Exploring the design space of highly-available distributed transactions

Alejandro Zlatko Tomsic

SCIENCES SORBONNE UNIVERSITÉ

THÈSE DE DOCTORAT DE
SORBONNE UNIVERSITÉ

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Alejandro Zlatko TOMSIC

Pour obtenir le grade de

DOCTEUR de SORBONNE 'UNIVERSITÉ

Sujet de la thèse :

Exploring the design space of highly-available distributed transactions

soutenue le 19 avril 2018

devant le jury composé de :

M. Marc SHAPIRO	Directeur de thèse
M. Vivien QUEMA	Rapporteur
M. Guillaume PIERRE	Rapporteur
M. Nuno PREGUIÇA	Examinateur
M. Pierre SENS	Examinateur
M. Lorenzo ALVISI	Examinateur
M. Sébastien MONNET	Examinateur

Abstract

The storage systems underlying today's large-scale cloud services handle a high volume of requests from users around the world. These services must provide fast response and an "always-on" experience. Failing to do so results in reduced user engagement, which directly impacts revenues. To fulfil these requirements, storage systems replicate data at multiple locations worldwide. Users minimise latency by connecting to their closest site and, in the case of site failures, users can connect to other healthy ones. Moreover, each location scatters data across a large number of servers. This way, each site can handle volumes of requests larger than what a single machine can handle.

Transactional guarantees simplify the development of applications that rely on storage systems. In particular, transactional isolation hides anomalous behaviour sourced in concurrency. However, in a distributed environment, their application can translate into users perceiving high latencies and service downtimes. This has lead to production stores —for instance, the ones underlying services such as Facebook and Amazon— to eschew isolation. This thesis studies how to enforce isolation in a cloud environment without affecting availability and responsiveness.

Our first contribution is Cure, a transactional protocol that ensures high level of semantics compatible with availability: Transactional Causal Consistency (TCC), an interactive transactional interface, and support for Convergent data types (CRDTs). TCC ensures there are no ordering anomalies, atomic multi-key updates and consistent-snapshot reads. Cure's interactive interface allows reading and updating objects in a single transaction. CRDTs expose a developer-friendly API and resolve concurrent updates safely, guaranteeing convergence and that no updates are lost. When compared to systems that eschew consistency and isolation, these guarantees minimise the anomalies caused by parallelism and distribution, thus facilitating the development of applications. Cure features a mechanism to make updates visible respecting causal order that incurs minimal overhead over systems that do not guarantee causal consistency. It relies on a novel metadata encoding to improve performance and progress with respect to state-of-art solutions. Experimentally, Cure is as scalable as a weakly-consistent protocol, even though it provides stronger semantics.

Transactional protocols like Cure simplify application development without compromising availability. Nevertheless, their transactional mechanisms exhibit latency overheads that have impeded their adoption at scale. Our second contribution is to explore how to implement distributed isolation with no extra delays with respect to a non-transactional system. In this quest, we find, quantify and demonstrate a three-way trade-off between read isolation, delay (latency), and data freshness. For our analysis, we identify a read-isolation property called Order-Preserving Visibility. Order-Preserving reads are weaker than Atomic reads, guaranteed by TCC and stronger models (e.g., Snapshot Isolation and Serialisability). They do not forbid a concurrency anomaly called Read Skew, which allows observing the updates of other transactions partially. On the positive side, like Atomic Visibility, Order-Preserving Visibility disallows reading uncommitted data and observing (e.g. causal) ordering anomalies.

The three-way trade-off between read isolation, delay (latency), and data freshness can be summarised as follows: *(i)* To guarantee reading data that is the most fresh without delay is possible only under a weakly-isolated mode, similar to that provided by the standard Read

Committed. *(ii)* Conversely, reads that enforce stronger isolation at minimal delay impose reading data from the past (not fresh). *(iii)* Minimal-delay Atomic reads force transactions to read updates that were committed and acknowledged in the past. *(iv)* On the contrary, minimal-delay Order-Preserving reads can read the updates of concurrent transactions. *(v)* Order-Preserving and Atomic reads at maximal freshness require mutually-exclusive read and write operations, which may block reads or writes indefinitely. These results hold independently of other features, such as update semantics (e.g., monotonically ordered or not) or data model (e.g., structured or unstructured).

Motivated by these results, we propose two isolation properties: TCC^{-} and PSI^{-}. They result from degrading the (Atomic) read guarantees of TCC and PSI to Order Preserving Visibility. Using the results of the trade-off, we use Cure, which's read algorithm sometimes blocks, to create three protocols which are latency optimal. AV maintains Cure's TCC guarantees by degrading freshness. The remaining two protocols improve freshness by weakening the isolation guarantees. OP provides TCC^{-}, and CV provides Read Committed Isolation, where reads enforce Committed Visibility.

The experimental evaluation of these protocols supports the theoretical results. All three protocols exhibit similar latency. The exception is Cure, which sometimes exhibits higher latency due to blocking. Regarding freshness, CV always reads the most up-to-date data. OP degrades freshness negligibly under all tested workloads, whereas under Cure and AV, the freshness degradation is severe.

Acknowledgements

This work would have not been possible without the many people who gave me their support and I worked with.

I am particularly thankful to my advisor, March Shapiro, for his guidance, help and support throughout this years. Also, for the exchanges that contributed to this work and my development. Finally, for opening many doors that included meeting the people and institutions we have collaborated with, many countries and two internships abroad.

I want to thank Inria, the LIP6 laboratory, and the Regal team, where I worked during this period. In particular, to thank Pierre Sens for opening its doors when I first entered as a masters intern, and for his kind support whenever needed. Also, to all the people I worked and shared time with: Mahsa Najafzadeh, Pierpaolo Cincilla, Marek Zawirski, Flavio Pileggi, Masoud Saeida, Ilyas Toumlilt, Santiago Alvarez, Vinh Tao, Michal Rudek, Michal Jabczyński, Rudyar Cortez, Dimitrios Vasilas, Sreeja Nair, Paolo Viotti, Johnathan Sid-Otmane, and Saalik Hatia.

I want to thank the SyncFree and LightKone European projects, who funded this PhD. Also, for the opportunity of collaborating with the involved institutions and their members. In particular, thanks to the coauthors of Part I of this work: Deepthi Devaki Akkoorath, Annette Bieniusa, and Nuno Preguiça for all our exchanges and time spent together.

I would like to thank the members of the jury, Sébastien Monnet, Vivien Quema, Guillaume Pierre and Roberto Alvisi for taking the time to read and review this manuscript, as well as for taking the time to travel to assist the defence of this thesis.

I am grateful to Sebastian Burckhardt and Phil Bernstein, who I had the opportunity to work with during the rich experience I had during my internship. Our daily interactions helped me learn about different aspects of programming models and databases, as well as a different way of working.

I am grateful to the people I collaborated with and had the chance to become friends with. Thanks Zhongmiao Li for all the discussions and laughs exchanged during these years. Also, to Tyler Crain for your help and exchanges during the first years of this PhD, for being always open to actively contributing to the work in this thesis, and the moments we exchanged outside the laboratory. Thanks to Christopher "workshop-brother" Meiklejohn, who I had the chance to (endlessly) discuss many aspects of this work and life with, and being a friend during the conference trips and internships. I am specially thankful to Manuel Bravo, who I was lucky to

meet when I first disembarked in Europe. Thank you for our endless exchanges, your contributions to this work, your strong support both in the professional and the personal, and offering me shelter whenever needed.

I want to thank my parents, Daniel and María del Rocío, and my siblings, Matías and Rocío. Thank you for your love and support, and for giving me wings to follow my path. Also, I want to thank my close friends for their care and support during this period, Ignacio Carreras, Joseba Dalmau, Ignacio Arch, Lourdes Ferro, Mudit Verma, Leandro Macchi, Analía Ramos and Julián Najles.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Preliminaries

Chapter 1

Introduction

Large-scale web services rely on highly-distributed, highly-parallel deployments to handle large load and volumes of data. For instance, "*Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world* [43]", and Tao, the data store storing Facebook's social graph "*runs on thousands of geographically-distributed machines, provides access to many petabytes of data, and can process a billion reads and millions of writes each second* [33]".

These services must serve requests in a timely fashion and provide an always-on experience. Response times directly affect revenues [44, 76] and, as Amazon has reported, "*even the slightest service outage has significant financial consequences and impacts customer trust* [43]". To reduce response times and tolerate failures, they employ geo-replication: deploying replicas of the application logic and state at multiple data centers worldwide. Users minimise latency by connecting to their closest site and, in the presence of failures that render a data centre unavailable, they can fail-over to other available ones. At each data centre (or replica), the application logic is deployed at multiple front-end servers, and the state is partitioned across multiple storage servers. This way, each replica can serve a volume of requests and store amounts of data beyond what a single machine can handle.

It is well known that, in this scenario, a system design must chose between simplicity of application development, and responsiveness and availability:

- Network partitions (P) and high latencies are unavoidable in long-distance and inter-continental network links. By the CAP theorem [50], a geo-replicated system must then choose between high availability (A) and strong consistency (C); ensuring both desirable properties simultaneously is impossible. Choosing strong consistency simplifies the task of developing the application logic, as it hides the complexity of geo-replication by keeping data-centers synchronised at all times. Nevertheless, it exposes users to the high-latencies and downtimes of wide-area network links. On the contrary, a design can chose to foster responsiveness and availability by serving user requests entirely from their closest

site, avoiding the downsides of cross-data-centre communication, and by synchronising data-centers lazily [55], yet this exposes concurrency, which renders application-logic development complex and error prone [25].

- Consistently reading and updating data scattered across machines requires implementing distributed transactions that enforce atomicity, the all or nothing property for updates, and read isolation, which ensures, for instance, all updates created atomically are observed simultaneously [24], the absence of order-related inconsistencies [60] and other anomalies sourced in concurrency [19]. Distributed transactions can hide the complexity of distribution from the application, but often incur communication overheads and blocking scenarios that directly impact latency [15, 63].

The above has motivated many latency-constrained production designs to eschew consistency and adopt fast multi-object operations with no transactional guarantees, thus exposing application developers, and sometimes users [25, 62], to anomalies sourced in distribution and replication. Examples include Facebook's Tao [33], LinkedIn's Espresso [73], Yahoo's PNUTS [39], and Amazon's Dynamo [43]

In this thesis, we study the design space of distributed and geo-replicated storage providing transactional semantics while guaranteeing availability and responsiveness similar to systems that, like the above-mentioned, implement no consistency or transactions whatsoever.

1.1 Contributions

In the first part, we introduce we present Cure[1], a transactional protocol that offers clear semantics that remain compatible with high-availability and low latency. Cure implements Transactional Causal Consistency (TCC) and offers support for conflict-free replicated data types (CRDTs). Cure implements these guarantees while achieving performance similar to weaker semantics.

In the second part, we analyse how to design distributed transactional protocols that do not exhibit extra delays with respect to non-transactional systems. We demonstrate a three-way trade-off between read isolation, delay (latency), and data freshness (the recency of the values a transaction reads). We use the results of the trade-off to modify Cure, which exhibits blocking scenarios, to derive novel isolation levels, and protocols with no extra delays.

1.1.1 Part I - Cure: Strong semantics meets high availability and low-latency

To alleviate the ease-of-development vs. performance problem, recent work has focused on enhancing AP designs with stronger semantics [60, 61, 81]. Cure is our contribution in this

[1]Cure is the transactional core and provides the base guarantees of Antidote DB [4], a database project that aims at providing applications with consistent storage exhibiting the minimal synchronisation required to respect their invariants. During this PhD, I've been an active contributor to Antidote's development.

direction. In comparison to previous available and low-latency protocols, it guarantees that *(i)* if one update happens before another, they will be observed in the same order, *(ii)* replicas converge to the same state under concurrent conflicting updates, *(iii)* support for high-level replicated data types (CRDTs) and not just registers with a last-writer wins policy, and *(iv)* transactions, ensuring that multiple keys (objects) are read and updated consistently.

Taken together, the above guarantees provide clear and strong semantics to developers. In fact, their combination equip Cure with the strongest semantics ever provided by an always-available data store. Cure implements these guarantees efficiently —it makes causally-ordered updates visible at remote sites fast while minimising metadata overhead. It achieves performance close to a weakly-consistent protocol, and outperforms other state-of-art systems.

The contributions of this part are the following:

- a novel programming model providing causally consistent interactive transactions with high-level, confluent data types (Section 5.1);
- a high-performance protocol, supporting this programming model for geo-replicated data stores (Section 6);
- a comprehensive evaluation, comparing our approach to state-of-the-art protocols (Section 7), which shows that Cure is able to achieve scalability and latency similar to protocols with weaker semantics.

1.1.2 Part II - The three-way trade-off for transactional reads

Systems like Cure provide clear semantics, high performance, and remain available under partition. Nevertheless, existing implementations exhibit delays that have impeded their adoption at scale [15]. In Part II of the thesis, we study how to build transactional protocols designed to incur no extra delay with respect to a non-transactional system. In this quest, we find that achieving minimal-relay reads imposes a trade-off between the freshness of the values transactions can read, and the level of isolation they enforce.

We consider three levels of read isolation: The weakest, Committed Visibility, ensures reading committed data (i.e., the read guarantees of Read Committed Isolation [19]). The strongest, Atomic Visibility, is summarises the read properties of many existing isolation levels, including TCC, Snapshot Isolation [27] and Serialisable Isolation [19]. When compared to Committed Visibility, atomic visibility further guarantees that reads i) do not observe ordering anomalies, given an order of updates. updates created atomically are observed by other readers respecting atomicity, and ii) do not observe ordering anomalies, given an order of updates. We identify the intermediate Order-Preserving Visibility. When compared to Atomic visibility, reads do not observe other transaction's updates atomically (an anomaly called Read Skew [27]).

We demonstrate the three-way trade-off between read isolation, delay, and data freshness, which we summarise as follows.

- Under Atomic Visibility it is possible to read with no extra delay, but then the freshest data is not accessible, only data that was stable (written and acknowledged) before the transaction started.
- Minimal-delay Order-Preserving Visibility improves freshness significantly over Atomic by allowing reading concurrent updates.
- If, on the other hand, the application requires the freshest data, under either Atomic or Order-Preserving Visibility, this is possible only if reads and writes are mutually exclusive, i.e., either might be delayed indefinitely by the other.
- The only model that allows transactions to access the freshest data with no extra delay is Committed Visibility.

Motivated by the results of the trade-off, we (i) propose isolation levels that result from combining Order-Preserving reads with different and update/commit semantics: TCC$^-$ and PSI$^-$. They result from degrading the (Atomic) read guarantees of TCC and Parallel Snapshot Isolation (PSI) [81] to Order Preserving. (ii) Moreover, use the results of the trade-off to drive protocol design. We modify Cure, which exhibits delays, to provide three protocols, all ensuring minimal delay. AV maintains Cure's TCC guarantees by degrading freshness. The other two improve freshness by weakening the isolation guarantees: OP provides TCC$^-$, and CV provides Read Committed Isolation, where reads enforce Committed Visibility.

Experimentally, as expected, the three protocols exhibit similar latency. Our protocol for Committed Visibility always observes the most recent data, whereas freshness degrades negligibly for Order-Preserving reads, and the degradation is severe under Atomic Visibility.

Chapter 2

System Model

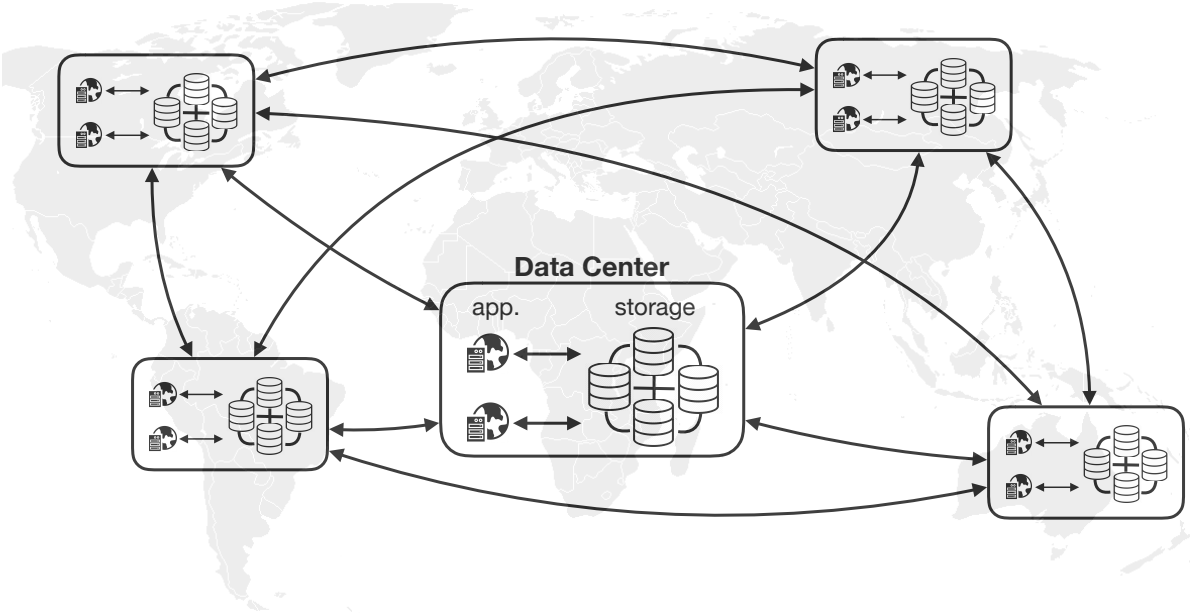


Figure 2.1: Typical Cloud Service Deployment at multiple interconnected data centres worldwide. Each data centre consists of front-end servers hosting the application logic and storage servers handling application data. This supports a high level of parallelism.

Cloud services rely on highly-parallel geo-distributed architectures handle requests of millions of users worldwide. For instance, "Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centres around the world [43]" and Tao, the data store storing Facebook's social graph "runs on thousands of geographically-distributed machines, provides access to many petabytes of data, and can process a billion reads and millions of writes each second [33]". Moreover, these services must reply to each user request fast, and must remain available in the presence of faults that render a data centre unavailable.

In this chapter, we introduce the architecture that we target throughout this work, and discuss the complexity of building transactional storage atop.

2.1 Cloud Service Architecture

Figure 2.1 shows a typical configuration, where a service is deployed at many data centres geo-distributed for fault-tolerance, and providing fast access to users (who can minimise latency by connecting to their closest site). To simplify reasoning, we assume that each data centre (or replica) stores a full replica of the service[1]. A data centre comprises a (possibly large) number of servers to fulfil processing and storage demands beyond what a single machine can handle. The service is configured in an application-logic and a storage tier.

- The ***application-logic tier*** (or front end) executes the service's business logic, and handles end user requests. When handling a request, the application-logic reads and updates data of the storage tier. Each application has its own set of rules that determine its correctness, called invariants. E.g., a banking system might ensure that a certain kind of account can never go exhibit a negative balance. The application logic must ensure that these invariants are preserved. In this thesis, we do not address application-logic mechanisms, including those that decide which data to access for read or write on behalf of a particular request.

- The ***storage tier*** (or storage backend) stores and manages access to application data by handling requests from the front-end tier. It exposes, to the application-logic tier, an interface for reading an updating data. *Transactional* storage systems allow multiple reads and/or writes to be expressed as a *transaction*, a group of operations that the storage system treats efficiently. For instance, a system can issue the operations in a transaction in parallel. Moreover, transactions provide a number of desirable semantics including atomicity and isolation (Section 3). Intuitively, the stronger the guarantees the storage tier provides, the easier it is to ensure application invariants and, therefore, the simpler the task of developing the application logic. If storage disallows concurrent modifications to the same data items, e.g.., concurrent withdrawals to the same account are impossible, it suffices for a developer to check the balance after each withdrawal/transfer to guarantee correctness. On the contrary, this check is insufficient over storage relaxing this guarantee.

This thesis addresses some of the trade-offs between a transactional storage system's performance and availability, and its guarantees. In the following sections, we elaborate on storage guarantees, performance, and the tension between them in our highly parallel and distributed model.

[1] In this work, we do not consider partial geo-replication, where each replica stores a subset of the application state.

2.2 Tight Latency and Availability requirements

2.2.1 The effects of latency

In this section, we summarise evidence of the negative effects of users perceiving high latencies.

Shopzilla, a shopping website, reported that a 5 second speed up (from around 7 to 2 seconds) to execute a purchase resulted in a 25% increase in page views, a 7-12% increase in revenue, a 50% reduction in hardware costs and a 120% increase in traffic from Google [44]. Amazon finds that every 100ms of latency costs them 1% in sales [76]. Google finds that adding 500 milliseconds to search page generation time decreases traffic by 20% [59].

Wall Street traders place orders of a small amount, and erase them after less than 500 ms. This is done to observe how slower traders react. This way, they obtain a competitive advantage over their competitors. "High-frequency traders generated about $21 billion in profits using this method in 2009" [11].

Psychological studies show that slow responses make users experience increased frustration [36]. Moreover, users tend to perceive slow websites as not trustworthy [49] and poor quality [31]. On the contrary, fast websites are perceived to be more interesting [74] and attractive [80].

2.2.2 The effects of downtimes

In this section, we summarise evidence of the negative effects of service downtimes. A 2004 study found that the tolerable wait time on non-working links without feedback peaked at between 5 to 8 seconds [68]. Amazon has reported that "even the slightest service outage has significant financial consequences and impacts customer trust [43]". When the Amazon.com site went down for approximately 49 minutes in January of 2013, it cost the company an estimated $4 million in lost sales. Another outage in August of the same year lasted 30 minutes and cost an estimated of almost $ 2 million. Google's five-minute outage in 2013 caused an estimated loss of $500.000 [8].

Because of the effects of high latencies and downtimes, system designs must foster latency and availability. As we will see in the following section, this comes at the cost of ease of programming.

2.3 Geo-distribution and the CAP theorem

Cloud services rely on geo-distribution to minimise user-perceived latencies and to maximise availability: a user connects to his closest data centre, thus avoiding wide-area network delays. In case of full-data-centre failures, he can be redirected to a healthy data centre. Nevertheless, this architecture poses a design choice to these services, known as the CAP theorem [50]. Long-distance inter-data-centre network links exhibit latencies in the order of the tens to hundreds of milliseconds, which are orders of magnitude higher than their under-millisecond intra-data-centre counterparts. Moreover, as network partitions between data centres do occur in production

systems and are more complex to handle than those between co-located servers [21], geo-replicated systems must be designed with network partitions (**P**) in mind. This forces a choice between (low-latency, weakly-consistent) highly-available (**AP**) and (high-latency, unavailable-under-partition) strongly-consistent (**CP**) designs: ensuring both strong consistency and high availability under network partitions is impossible.

2.3.1 CP designs

A strongly-consistent design simplifies the task of developing the application logic. It provides the abstraction of a single sequential system as it hides the complexity of replication by keeping replicas synchronised at all times. Nevertheless, it exposes users to the high latencies and downtimes of the network links between replicas as operations have to be synchronised across data centres before finishing execution.

2.3.2 AP designs

A highly-available design provides an "always-on" experience and excellent responsiveness by synchronising replicas lazily, out of the critical path of an operation [55]. Users can execute operations entirely at a single data centre, avoiding the need to wait for data centres to synchronise, which happens in the background, after replying to the client. Nevertheless, it exposes concurrency issues that render application-logic development hard and error prone [25]. We introduce such issues in Section 3.3.3.

Motivated by the tight latency and availability requirements of these systems, in this work, we focus mainly on AP designs.

In the following sections, we introduce the guarantees provided by many AP and CP isolation models. We explain how these guarantees affect performance and ease of programming.

Chapter 3

Storage Semantics

Traditionally, transactional storage provides a way to access data in an efficient and programmatically-simple fashion —for instance, through a query language such as SQL [19]. In this chapter, we introduce background on these system's properties.

3.1 Transactions - ACID properties

***A*tomicity** guarantees that all the effects of a transactions are in the store, or none is. E.g., that executing a transfer in a bank application both withdraws money from a source account and makes the corresponding deposit to a destination account, or does not execute either action;

***C*orrectness/Safety** ensures that each transaction individually updates state respecting application invariants. E.g., that the balance of a bank account is never negative;

***I*solation** establishes which intermediate states (if any) transactions can observe of other transactions. In the bank transfer example, strong isolation ensures that, if balance inquiries to the involved accounts are done concurrently with the transfer, these inquiries do not observe a state where money was withdrawn from the origin and not deposited to the destination (or vice-versa);

***D*urability** ensures that when an update transaction (i.e., a transaction that updates data) is acknowledged by the system, its effects will be visible to other transactions.

In this work, we focus in those aspects of correctness (C) related to storage: atomicity and isolation. Moreover, we do not focus on durability, which we take for granted throughout this thesis. Finally, we do not address the issues of porting query languages and query-processing mechanisms to large-scale deployments (a recently-started trend [10, 12]).

3.1.1 Moving away from, back to ACID

Transactional semantics simplify the task of programming the application logic: the ACID properties eliminate the concerns of handling concurrent access to data consistently. Nevertheless, the advent of distributed and replicated architectures has pushed designs, called NoSQL, to completely eschew query languages and transactional isolation [33, 37, 39, 43, 73]. The reason behind this trend is availability and latency. Isolation mechanisms were originally designed for single-machine architectures, and porting them as is to a cloud environment results in storage that is slow and unavailable under certain kinds of failures not present in the single-machine environment (e.g., network partitions between data centres). Therefore, developing the application logic on top of these stores is a hard task.

Recently, the complexity of developing applications over stores with no transactional isolation has motivated the development of distributed transactional isolation. This has resulted in production systems and research prototypes providing a wide variety of guarantees for cloud environments [7, 40, 60, 70, 71, 75, 81].

We focus, in the following sections, in the inherent trade-offs of implementing atomicity and isolation in cloud deployments and recently proposed isolation models.

3.2 Atomicity of Updates (or All-or-Nothing)

All-or-Nothing ensures that, at any point in time, either all of a transaction's writes are in the store, or none of them is. They are instrumental for ensuring state transitions consistently with respect to certain invariants. Examples include foreign key constrains to represent relationships between data records (e.g., the symmetry of the friendship or the like relationships in a social network application [33]: if Bob belongs to Alice's friends, then Alice belongs to Bob's and, if Bob likes a photo, the photo is liked by Bob), secondary indexing and materialised view maintenance (e.g., keeping a list of comments of a certain post and a comments count can be done by updating, with each comment creation, the comments count instead of computing this count on reads) [24]. Moreover, all-or-nothing updates simplify rolling back inconsistent intermediate state of failed transactions. Consider, for instance, a bank application where a transfer withdraws money from an account and deposits it into another. If the withdrawal succeeds but not the deposit, programmers using storage lacking atomic updates need to develop mechanism to detect and roll back the withdrawal, such as a compensating deposit.

In transactional storage, atomically-updating data in parallel (or distributed) environments is achieved by an atomic commitment protocol such as Two-Phase Commit [57], where all updated entities must agree on applying their individual updates, before the transaction is effectively committed.

In the following section, we present existing read isolation properties. Unless stated otherwise, they assume update atomicity.

3.3 Transactional Isolation Levels (Consistency Criteria)

The stronger the level of isolation a system implements, the more its behaviour resembles that of a sequential system. However, implementing strong isolation in a highly-parallel setting requires concurrency control, i.e., techniques that limit the possible interleaving of a transaction's operations. This introduces an overhead over a transaction's execution. Weakening isolation exposes anomalous effects of concurrency, called anomalies. The presence of anomalies increases the complexity of developing correct applications. On the positive side, reducing concurrency control boosts performance by imposing fewer restrictions on how operations interleave, which enables a variety of optimisations.

A large body of research, both from the parallel and the distributed programming communities, has focused on proposing isolation levels (also called consistency criteria) along the design space created by the isolation-performance trade-off. In this section, we introduce isolation levels and concurrency control techniques. In Chapter 12, we propose new isolation levels and, in Chapters 5 and 13, new implementations thereof. In Chapter 16, we elaborate on how existing systems and research prototypes implement isolation. A list of many commercial database systems and their default and maximum offered isolation level can be found elsewhere [23].

3.3.1 Notation

The application state is composed of objects (or items), noted $x, y, \ldots$. Each read or update (or write) operation acts on a single item. A read operation returns the object's value, and an update operation modifies its value. A transaction T is a finite sequence of read and write operations followed by a terminating operation, commit or abort. A transaction that commits applies its updates, making them visible to other transactions[1]. When a transaction aborts, all its updates are discarded.

3.3.2 Concurrency control

Isolation is achieved through concurrency control mechanisms. There are two main techniques:

3.3.2.1 Lock-based concurrency control

This technique relies on locking objects prior to reading and/or modifying them —depending on the isolation property the mechanism ensure. This technique is called pessimistic concurrency control, as transactions lock objects even when other concurrently-executing transactions might not need to access them.

[1]We do not consider, throughout this document, any isolation property or transactional system that allows transactions to observe uncommitted data such as, for instance, the ANSI Read Uncommitted [19].

3.3.2.2 Multi-version concurrency control (MVCC)

MVCC is a form of optimistic concurrency control (OCC) that relies on keeping multiple versions of each object. Transactions are allowed to execute optimistically under the assumption that no concurrency issues will happen, and only checked for concurrency issues at the end.

Optimistic Execution. Under MVCC, transactions read object versions from a database snapshot. A snapshot represents a view of the state of the store composed by a version of each object. Each isolation property defines the rules that object versions must satisfy to belong to a snapshot. For instance, a requirement of strong isolation models (e.g., Serialisability and Snapshot Isolation) is that all atomically-created updates are in a snapshot, or none is. This is not a requirement of weak isolation (e.g., Read Committed and Read Uncommitted isolation).

Concurrency checks. At commit time, if necessary, the read and/or update operations of a transaction undergo checks to verify the transaction has not interleaved with other concurrent transactions in ways forbidden by the target isolation property. If the certification passes, the transaction commits. Otherwise, it aborts [28]. For instance, under Snapshot Isolation, a transaction reading an updating a certain object must certify that no other transaction has modified the same object since the object was read. In case a concurrent modification is detected, the transaction aborts (Section 3.3.4.3).

3.3.2.3 Choosing a technique.

MVCC does not incur the overheads of acquiring and removing locks. MVCC is useful to implement strong isolation when the levels of contention are low. High contention can cause interleaving transactions to access the same objects, which can lead to transactions being aborted and retried frequently. MVCC is also the only technique used to implement weak isolation where transactions do not require exclusive access to objects and, therefore, they never abort due to concurrency issues.

Lock-based concurrency control is suited for strong isolation, which requires exclusive access to objects. In particular, this technique is applied to workloads that exhibit high levels of contention, as it never causes a transaction to abort.

3.3.2.4 Mixing them.

Mixing optimistic and pessimistic techniques in a single system is possible and common practice. An implementation can, for instance, rely on locks for highly-contended objects and on MVCC for objects which are less contended.

3.3.3 Anomalies

The highest level of isolation, Strict Serialisability, provides an abstraction where each transaction executes alone in the system —thus not interfering with other transactions— in the same order they are received by the system [52, 69]. Therefore, running under the strongest isolation prevents observing concurrency issues. An *anomaly* is a (normally undesirable) effect that results from allowing multiple transactions to execute concurrently in a system that weakens isolation, i.e., which exposes operations or transactions to the intermediate states of others. Anomalies help defining isolation properties, as they exemplify undesirable situations that are possible under a particular model, which is useful for a programmer to take action at the application level when needed (an anomaly might be harmful to some applications and harmless to others). We first introduce concurrency anomalies, to later define isolation levels according to both undesirable anomalies and desirable properties they offer.

In what follows, we list the definition of anomalies, as they will be referred from the rest of this document. We suggest the reader to jump to Section 3.3.4, and to read each definition when necessary.

3.3.3.1 Dirty read.

A dirty read occurs when a transaction T_r reads an update made by a transaction T_u that has not yet committed. It could happen that T_u aborts, invalidating the update observed by T_r.
Example. Initially $x = 0$. T_u sets $x = 1$ and later $x = 2$ and commits (or aborts). Concurrently, T_r reads $x = 1$.

In this work, we do not consider this anomaly as its avoidance can be achieved trivially, for instance, by buffering the reads a transaction has already performed.

3.3.3.2 Non-repeatable read.

A non-repeatable read occurs when a transaction reads the same object twice, obtaining different results.
Example. Initially $x = 0$. T_u updates $x = 1$ and commits. Concurrently T_r reads $x = 0$ and then reads $x = 1$.

3.3.3.3 Lost update.

A lost update occurs when multiple transactions make updates concurrently to a common object, causing one transaction's updates to be lost, i.e., unobservable by other transactions.
Example. Initially $x = 0$. T_u reads $x = 0$, writes $x = 1$, and commits. Concurrently, T'_u reads $x = 0$, writes $x = 2$, and commits.

3.3.3.4 Write skew.

A write skew occurs when a transactions executing concurrently read an intersecting group of objects and make updates to disjoint objects belonging to that intersection read. None of these transactions observe the effects of the other(s). This anomaly is also known as short fork [81].
Example. Initially $x = y = 0$. T_u and T'_u read $x = y = 0$. Concurrently, T_u writes $x = 1$ and T'_u writes $y = 1$. Then, both transactions commit.

3.3.3.5 Non-monotonic snapshots.

Non-monotonic snapshots are observed when concurrent transactions make updates to different objects. After they both commit, transactions observe the effect of those transactions as occurring in different order. This anomaly is also known as long fork [81].
Example. Initially $x = y = 0$. T_u updates $x = 1$, and commits. Concurrently, T'_u writes $y = 1$, and commits. T_r and T'_r execute concurrently with T_u and T'_u. T_r reads $x = 1, y = 0$, and T'_r reads $x = 0, y = 1$.

3.3.3.6 Read Skew.

A transaction updates multiple objects. Other transactions reading a number of those objects observe some but not all the updates of the updating transaction [27]. This anomaly is also known as a fractured read [24].
Example. Initially $x = y = 0$. T_u updates $x = y = 1$, and commits. Concurrently, T_r reads $x = 1, y = 0$ (or $x = 0, y = 1$).

3.3.3.7 Real-time violation.

This anomaly occurs when two or more transactions execute in a certain order as witnessed by an external observer, and later transactions do not observe the effects of earlier ones. I.e., A transaction T_u performs one or more updates and commits. A transaction T_r, issued after T_u finished, reads objects updated by T_u but does not observe T_u's updates.
Example. Initially $x = 0$. T_u updates $x = 1$ and commits. Later, T_r reads $x = 0$.

3.3.3.8 Order Violation.

This anomaly occurs when two or more transactions execute in a some significant order established by the system, and a transaction performs a number of reads, observing gaps in that order.
Example. Initially $x = y = 0$. T_u updates $x = 1$ and commits. After, T'_u reads $x = 1$ (the update made by T_u), updates $y = 2$ and commits. T_r, running concurrently with T_u and T'_u, reads $x = 0, y = 2$.

3.3.4 CP (Strong) Isolation

A strongly consistent models is one that does not allow concurrent transactions to commit modifications to the same object, thus disallowing the lost update anomaly. Under geo-replication, this requires across-data-centre communication on the critical path of an update transaction. In this section, we introduce CP isolation models. Table 3.1 summaries the anomalies these models present.

3.3.4.1 Strict Serialisability (SS) - no anomalies

This is the strongest isolation property [69] and, therefore, the simplest to program against. It disallows all anomalies. It ensures that every transaction appears to execute at a single point in time between its beginning (or first operation), and its commit point. More precisely, SS ensures that the results of concurrently committed transactions is equivalent to a serial execution (i.e., one transaction after another) that respects the order of real time (a property called External Consistency [58]). Under SS, consistency is guaranteed also when communication happens outside the boundaries of the system. For instance, Alice makes a deposit into Bob's account and calls to let Bob know over phone. SS guarantees that Bob will observe the deposit in his account when checking his balance. This guarantee is not provided the other isolation models we consider.

3.3.4.2 Serialisability (S) - relaxing real-time ordering

S is very similar to SS. S ensures that the execution of committed transactions is equivalent to *some* serial execution of the same transactions, but not necessarily in real-time order. Therefore, it allows the occurrence of real-time violations.

The performance benefit of allowing real-time violations. Allowing real-time violations gives the system flexibility to order transactions in multiple ways, which allows for increased concurrency. Consider the following example. Initially $x = y = 0$. T_u updates $x = y = 1$ and commits. T_r reads x, y while T_u is committing (T_u has not finished). The system can chose to reply immediately with $x = y = 0$ (and does not need to wait for T_u to commit to reply with $x = y = 1$, as it should under SS), meaning that it can serialise T_r as happening before T_u, and allow two transactions to run concurrently instead of one.

3.3.4.3 Snapshot Isolation (SI) - removing serialisability checks from read operations

This isolation property was defined in terms of how a transaction should execute under a given implementation [27], which is based on MVCC. Under SI, a transaction T reads from a possibly-stale snapshot of the database (thus allowing real-time violations) and makes updates over that

view. When finishing execution, T aborts if a concurrent transaction has committed updates to some object updated by T since T's snapshot.

SI is one of the most widely-adopted isolation properties, and it is supported by most commercial database systems [23]. The rationale behind its design is that reads predominate in many workloads, and removing their serialisation checks improves the overall performance of the system significantly. However, removing serialisability checks from reads comes at the cost of exhibiting the write-skew (or short-fork) anomaly.

The benefit of removing serialisability checks from read operations. Serialisability checks can require blocking objects or extra messages to servers storing read objects, which can be particularly expensive under replication. By removing serialisability checks from reads, under SI, transactions execute faster and read-only transactions never abort.

3.3.4.4 Update Serialisability (US) - non-monotonic snapshots

US ensures that update transactions are serialisable, but relaxes isolation for read-only transactions, which can observe the non-monotonic snapshots (or long-fork) anomaly [**?**]. Similarly to SI, read-only transactions read from a consistent snapshot of the state, and do not undergo concurrency checks. Differently, both the reads and updates of update transactions undergo serialisability checks and, for this reason, US does not exhibit the write-skew anomaly.

The benefit of non-monotonic snapshots. By removing the requirement of monotonic snapshots, implementations can avoid synchronisation, particularly under geo-replication. To illustrate this, consider the following example execution in a geo-replicated system comprised of two data centres DC_1 and DC_2:

Initially $x = y = 0$ at both DC_1 and DC_2. T_u^1, running at DC_1, updates $x = 1$ and commits. Concurrently, T_u^2, running at DC_2, writes $y = 1$ and commits. By allowing non-monotonic snapshots, T_r^1 can execute at DC_1 after T_u^1 commits, and read $x = 1, y = 0$, while T_r^2 can execute at DC_2 after T_u^2 commits, and read $x = 0, y = 1$. T_r^1 and T_r^2 can proceed before DC_1 and DC_2 have exchanged the updates made by T_u^1 and T_u^2. This execution is not serialisable, as it is impossible to order T_u^1 with respect to T_u^2, or T_r^1 with respect to T_r^2 (T_r^2 has observed T_u^2's effects before those of T_u^1 and T_r^1 has observed T_u^1's effects before those of T_u^2).

To disallow this behaviour, the system must either establish an order between T_u^1 and T_u^2 before T_r^1 and T_r^2 read the values of x and y, or hide their updates from other transactions until such an order is established. The former alternative requires synchronising DC_1 and DC_2, which can slow down some of these transactions. The latter has two problems: (i) it lets T_r^1 and T_r^2 read stale data (with respect to what is already available at their sites of execution), and (ii) if T_u^1 and T_r^1 (or T_u^2 and T_r^2) were executed by the same client, this would violate the read-your-writes session guarantee (Section 3.4).

3.3.4.5 Parallel Snapshot Isolation (PSI)

PSI, also known as Non-Monotonic Snapshot Isolation [75], combines the relaxations made by SI and US over S, thus allowing both their anomalies (long fork and write skew)[81].

3.3.5 AP (Weak) Isolation

AP designs rank availability and performance over ease of programming. They further remove serialisability checks from updates, thus allowing concurrent updates to the same objects. The benefit is that update operations do not acquire locks or never abort due to serialisation checks, which allows to completely avoid synchronous operations under geo-replication. Therefore, AP models exhibit the lost update anomaly. To avoid coordination, the first distributed and replicated AP designs provided no isolation (NI) or update atomicity, allowing all possible anomalies [33, 37, 39, 43, 73]. Later research has proposed stronger isolation that remains available under partition. We introduce such models in this section. Table 3.1 summaries the anomalies presented by both CP and AP isolation models.

3.3.5.1 Transactional Causal Consistency (TCC)

Under TCC, updates respect causal order, which guarantees that if one update happens before another, they will be observed in the same order (Section 3.5.2.1). TCC guarantees a transactions reads committed data, from a snapshot that respects causal order and ensures a transaction does not read the updates of another transaction partially. Under geo replication, it is the strongest (so-far-proposed) isolation property that remains always available under partitions. A detailed definition of this model is provided in Section 5.1.

3.3.5.2 Read Atomic (RA)

RA is a model that ensures that transactions observe committed data, and that they do not observe the updates of other transactions partially. Therefore, RA only forbids transactions to observe uncommitted data and read skews [24].

3.3.5.3 Read Committed (RC and RC$^+$)

The standard Read Committed (RC) ANSI isolation level [19] only prevents transactions from reading uncommitted data. Therefore, all anomalies except dirty reads are observable. When compared to TCC, it allows causally-consistent snapshot violations and the Read-Skew anomaly. RC is sometimes specified in terms of a lock-based implementation that disallows the Lost-Update anomaly. Under replication, this requires synchronous updates (which turns RC into a CP model). Throughout this work, we call this variant **RC**$^+$.

	CP Models						AP Models			
Anomalies \ Isolation	**SS**	**S**	**US**	**SI**	**PSI**	**RC+**	**TCC**	**RA**	**RC**	**NI**
Dirty Read	x	x	x	x	x	x	x	x	x	-
Read Skew	x	x	x	x	x	-	x	x	-	-
Lost Update	x	x	x	x	x	x	-	-	-	-
Write Skew	x	x	x	-	-	-	-	-	-	-
Non-monotonic snapshots	x	x	-	x	-	-	-	-	-	-
Order violation	x	x	x	x	x	-	x	-	-	-
Real-time Violation	x	-	-	-	-	-	-	-	-	-

Table 3.1: Anomaly comparison of isolation levels

The benefit of allowing Order violations and read skews. To disallow these anomalies, existing APdesigns (i) keep multiple versions of each object, and (ii) apply concurrency control to transactions to disallow them observing these anomalies. Item i requires extra storage and processing. Under existing implementations, item ii affects latency.

3.3.5.4 No Isolation (NI)

NI permits all introduced anomalies. In particular, under NI it is possible to observe uncommitted data. NI is called Read Uncommitted (RU) in the ANSI standard [19].

3.3.6 Summary of anomalies allowed/disallowed by Isolation levels

Table 3.1 summaries the anomalies presented by the introduced CP and AP isolation models.

3.4 Session guarantees

Session guarantees provide a client with a view of the data store that is consistent with his own actions [84]. Their implementation is often independent from that of isolation levels.

Monotonic reads. This property requires that a client never observes state older than what it has observed during a previous operation. Consider the following example: Initially, $x = 0$. T_u updates $x = 1$. T_r reads $x = 1$. From this moment on, further transactions issued by the client that issued T_r must observe $x = 1$ or the result of subsequent updates.

Writes follow reads. This guarantee ensures that, if a client observes an update of a transaction T_u and subsequently performs updates in a transaction T'_u, any client will observe the effects of T'_u only if she can also observe those of T_u.

Monotonic writes. This property guarantees that a user's updates are applied by the system in the order they were submitted.

Read your writes. This property guarantees that if a client performs certain updates, further reads issued by the same client will observe their effects.

3.5 Single-object Consistency and Isolation

We present single-object consistency models. They can be classified into AP and CP.

3.5.1 CP Consistency

3.5.1.1 Linearisability

This property is equivalent to Strict Serialisability (Section 3.3.4) for single objects [52]. It guarantees that the operations on an object are executed sequentially in the order they were submitted. Strict Serialisability ensures each object is linearisable.

3.5.2 AP Consistency

3.5.2.1 Causal consistency

To define this model, we first define causal order, characterised by the happens-before relation $\rightsquigarrow$ [56].

Definition 1 (Happens Before)**.** For two operations a and b, we say a happens before b or, equivalently, $a \rightsquigarrow b$, if any of the following conditions hold:

- *Thread-of-execution*: a and b are executed by the same thread (or client), and a is ordered by the thread before b.
- *Reads-from*: a performs an update, b reads the values written by a.
- *Transitivity*: There exists c such that, if $a \rightsquigarrow c$ and $c \rightsquigarrow b$, then $a \rightsquigarrow b$.

Causal consistency requires that clients observe updates in causal order. Therefore, if updates a to an object A and b to an object B happened in the order $a \rightsquigarrow b$ and a client that reads B observes the effects of b, when she later attempts to read A, her read must observe the effects of a. We say that, if $a \rightsquigarrow b$, then a is a causal dependency of b. This property is similar to TCC (Section 3.3.5) for single objects [13].

A key ingredient in consistency criteria. Causal order is a key ingredient of TCC, PSI and US. Systems implementing those models include a mechanism to causally order updates.

Mechanisms to ensure causal consistency in cloud environments. In a replicated setting, causally-related updates can arrive out of order to remote replicas. In this situation, a receiving replica must ensure that it makes these updates visible to clients respecting causal order. The situation is more complex when a replica is partitioned, as a given update and its

causal dependencies might be stored at different servers, requiring extra communication. There have been four mechanisms proposed by the literature under this setting: (i) explicit dependency check messages, (ii) dependency stabilisation, (iii) dependency dissemination trees, and (iv) no dependency checking, that we also call unavailable causal consistency.

To explain each of these mechanisms, consider a sample cloud deployment that stores two objects A and B, comprised of two replicas R_1 and R_2, each comprised by two partitions, storing one object each: $p_1^A, p_1^B \subset R_1$, $p_2^A, p_2^B \subset R_2$. Now consider an execution that creates, at R_1, updates a to A and b to B, such that $a \rightsquigarrow b$. If the system provides causal consistency, then any client that reads B and observes b, must observe a (and possibly updates to A causally depending on a) when later reading A.

Explicit Dependency Check Messages. Upon arrival of an update, a partition receiving communicates with each partition storing the update's dependencies to verify that these dependencies have been applied. In our example, when R_2 receives update b from R_1, p_2^B will send a message to p_2^A asking if it has applied a. p_2^A replies to p_2^B only after receiving and applying a (which, in turn, might require similar checks). p_2^B makes b *visible* to read operations only after receiving this response.

Dependency Stabilisation The goal of this mechanism is to reduce the number of messages required to make an update visible respecting causal order, with respect to explicit dependency check messages. A partition groups, and periodically broadcasts, information regarding the updates it have recently received from sibling partitions at other replicas. Upon receiving such information from all the partitions of its local data centre, a partition can compute locally which of its received updates is ready to be made visible to readers. The number of messages exchanged between servers remains constant with the number of updates. Its downside is that updates take potentially more time to be made visible or, equivalently, *update-visibility latency* is larger.

In our example, p_2^A and p_2^B periodically exchange messages informing each other of the updates each has received from p_1^A and p_1^B, respectively. Consider, for instance, p_2^B receives b before p_2^A receives a. p_2^B buffers b until one of the periodic messages from p_2^A informs it has applied a.

Dependency Dissemination Trees This technique consists of building a tree that takes care of delivering updates to other replicas in causal order. Each replica acts as a node in the tree and submits its updates to it.

Dependency dissemination trees are particularly useful for edge networks, where the number of replicas is large, and where replicas do not replicate the entire state (i.e., under partial replication). If the tree is built carefully taken into consideration across-replica latencies, it exhibits high throughput and low update-visibility latency. The downside of this mechanism is that it requires each replica to totally order its updates and submit them to the tree respecting

this order. Moreover, each replica subscribes to the tree through a single point, which delivers a stream of updates. This single point might become a bottleneck in large data centres. Other techniques do not pose this restriction. They allow partitions to exchange updates independently.

In our example, p_1^A and p_1^B submit updates to the entry point of tree at $R1$ in the order ab. The tree delivers these updates to $R2$ through an entry point, that sends a to p_2^A, and b to p_2^B, respecting this order.

No dependency Checking (or unavailable causal consistency). All mechanisms introduced above guarantee high availability by applying updates in causal order (which includes enforcing the monotonic-writes session guarantee). However, it is also possible to implement this model without guaranteeing high availability, by applying updates arriving from remote replicas immediately, and performing dependency checks at read time. This mechanism does not guarantee high availability, as it allows an update that arrives before its causal dependencies to be read, while its dependencies may not arrive due to a network partition.

In our example, consider that p_2^B receives b before p_2^A receives a. p_2^B applies b immediately, making it visible to readers. A client reads B, observing b and later reads A. If p_2^A has not received a, the client must block, possibly indefinitely, until the arrival of a.

In Section 16.1, we will discuss how existing systems enforce causal order.

3.5.2.2 Eventual Consistency

Eventual Consistency ensures that replicas that have received the same updates converge to the same state independently of the order in which they process them.

3.5.2.3 Ensuring Convergence

Concurrent operations are not ordered under AP consistency. If two concurrent operations update the same object, then they can lead to a conflict. Under replication, this can lead to divergent replicas and losing updates. Ensuring convergence requires the adoption of a mechanism to resolve conflicting operations. Examples include exposing the conflict to be resolved by the application, or relying on conflict-free data types (CRDTs) [78], such as registers with a the last-writer wins rule [85].

The last-writer-wins (LWW) rule. In the presence of concurrent conflicting updates, data types implementing LWW keep the update that occurred "last". Each update has a unique identifier taken from a totally-ordered space. When a replica applies concurrent updates, it picks the one with the highest identifier and discards the remaining ones. As each replica can perform the same action, independently, without communicating to other replicas, this mechanism guarantees Eventual Consistency. This technique achieves convergence, but does not prevent the Lost-Update anomaly (in the presence of concurrent conflicting updates, some may be discarded).

Convergent and Commutative Data Types (CRDTs). CRDTs are data types such as registers counters, sets, lists, tables and maps that implement a mechanism to resolve concurrent conflicting updates automatically.

CRDTs guarantee convergence by ensuring *all operations commute*: applying a number of operations to a CRDT in an initial state will converge to the same final state independently of the order in which those operations are delivered [78]. The simplest example is a counter that exposes an interface consisting of an $increment$ and a $decrement$ operation. A counter c with a given initial value will exhibit the same final value if one applied $c.increment$ followed by $c.decrement$ or the same operations in the inverse order (the same would happen with two increments or two decrements).

Some CRDTs disallow the lost update anomaly, as they apply all conflicting operations. Consider, for instance, a two-replica deployment where a counter's initial value at both sites is $c = 1$ and two users increment the value of the counter at different replicas concurrently. A CRDT counter will guarantee that both increments are applied at both sites when the system synchronises. There are a number of ways to implement this behaviour. We introduce two of them.

Operation-based implementation. Under an operation-based implementation, replicas exchange operations. These operations encode the state over which they must be applied. Under replication, an operation-based implementation of a CRDT requires updates to be applied in causal order and exactly-once at remote replicas.

Causal order ensures that, at a remote replica, an update is applied after the updates that were present at its source. As an illustrative example, imagine an empty CRDT set s in a replicated system where one replica executes $s.add(A)$ followed by $s.remove(A)$. If the remove operation arrives to the other replica before the add operation, the resulting state at one replica could show an incorrect state where $A \in s$.

Exactly-once delivery is required by operations that are not idempotent. For instance, applying more than once the same increment to a counter would lead to an incorrect value.

State-based implementation. The state-based approach relies on state that encodes the information of how to handle remote updates. A state-based CRDT implementation does not require exactly-once delivery nor causal order from the system for correctness. In the case of replication, this state is exchanged among replicas. When receiving state from a replica, each replica locally applies a merge function over its local and received states to compute its current state.

Observed-remove set example. We compare an operation and a state-based implementation of a set. A set exposes two operations: $add(E)$ adds element E to the set, and $remove(E)$ removes element E from the set. Two operations do not commute: add and remove over the same

element in the set. There are many possible policies to resolve this situation. We present an implementation with intuitive semantics, called the *observed-remove set*, or OR-Set, where a $remove(A)$ operation removes the "observed" instances of A. We chose the OR-Set as it is used in the evaluation of Cure (Part I). OR-Sets are also called Add-Win-Sets as, in the presence of conflicting concurrent $add(A)$ and $remove(A)$ operations, A will remain in the set since the remove operation has not observed the concurrent add operation.

Sketch of a state-based implementation. Each add operation is internally assigned a unique identifier, and each remove operation, a list of the identifiers of the add operations observed by the remove operation. The state of the set results from the add operations which have not been marked as removed.

For instance, consider the following sample execution. Initially, set $s = \emptyset$. The following operations are applied. $s.add(A)$ followed by $s.add(A)$. The internal representation of s after those updates will be: $s = \{(A, uid1), (A, uid2)\}$. A later $s.remove(A)$ will result in $s = \{(A^*, [uid1, uid2])\}$, where $(A^*, [uid1, uid2])$ is a "tombstone" indicating that a remove operation on A has removed the (observed) $add(A)$ operations identified by $uid1$ and $uid2$.

This implementation presents two drawbacks: (i) remove operations leave tombstones that cannot be garbage collected (as exactly-once delivery is not assumed,[2] and (ii) under replication, the entire state has to be exchanged by replicas.[3]

Sketch of an operation-based implementation. We sketch a possible operation-based Set implementation. We assume that each operation is augmented with an unique identifier. Identifiers respect causal order. As we will see, this simplifies garbage collection. Moreover, operation exchanges among replicas are lighter than state exchanges.

We illustrate the algorithm with the same example execution of $s.add(A)$ followed by $s.add(A)$ over $s = \emptyset$. Internally, the implementation will assign $uid1$ to the first operation, and $uid2$ to the second, where $uid1 \rightsquigarrow uid2$. The internal representation of the s after those updates will be: $s = \{(A, uid2)\}$: only the causally-latest add needs to be recorded and, in the replicated case where updates are sourced at the same replica, transferred.

A later $s.remove(A)$ will be assigned $uid3 : uid2 \rightsquigarrow uid3$. This will result in the following internal representation: $s = \emptyset$: as the remove operation has a causally-after id, all add operations to A with smaller identifiers can be removed.

[2] An optimised implementation removes this problem [30].

[3] There are some optimised state-based implementations that reduce the size of the state exchanged [17].

Part I

Cure: strong semantics meets high availability and low latency

Chapter 4

Introduction to Part I

Many cloud services are layered over a high-performance distributed data store running at a number of data centers (DC) worldwide. Geo-replication across several DCs saves users wide-area-network latencies and partitions, and DC downtimes. As presented in Section 2.2, this is of paramount importance for such systems.

Traditional CP databases provide ACID guarantees and a high-level SQL interface, but lose availability. In contrast, AP databases are highly available and bring significant performance benefits. However, they expose application developers to inconsistency anomalies, and most provide only low-level key-value interface (Section 2.3).

To alleviate this problem, recent work has focused on enhancing AP designs with stronger semantics [60, 61, 81]. In this part of this work, we present Cure, our contribution in this direction. While providing availability and performance, Cure provides *(i)* Transactional Causal Consistency (TCC), i.e., causal consistency, ensuring that if one update happens before another, they will be observed in the same order, *(ii)* support for operation-based replicated data types (CRDTs) such as counters, sets, tables and sequences, with intuitive semantics and guaranteed convergence in the presence of concurrent updates and partial failures, and *(iii)* general transactions, ensuring that multiple keys (objects) are both read and written consistently.

Causal consistency (CC) represents a sweet spot in the availability-consistency trade-off [14, 60]. It is the strongest model compatible with availability for individual operations [20]. Since it ensures causal consistency (introduced in Section 3.5), it is easier to reason about for programmers and users. Consider, for instance, a user who posts a new photo to her social network profile, then comments on the photo on her wall. Without causal consistency, a user might observe the comment but not be able to see the photo. To avoid the anomaly, this requires extra programming effort at the application level.

CRDTs are developer-friendly high-level data types that have rich semantics (Section 3.5.2.3). Operations on CRDTs are not only register-like assignments, but methods corresponding to the CRDT object's type. CRDTs ensure that replicas eventually converge to the same state despite concurrent conflicting updates. For guaranteeing convergence, previous causal+ consistent

systems [18, 45, 47, 60, 61] adopt the last-writer-wins rule, where the update that occurs "last" overwrites the previous ones. Cure provides support for operation-based CRDTs. For instance, the Bet365 developers report that using Set CRDTs changed their life, freeing them from low-level detail and from having to compensate for concurrency anomalies [64].

Performing multiple operations in a transaction enables the application to maintain relations between multiple objects. *AP isolation* eschews traditional strong isolation properties, which require synchronisation, in favour of availability and low latency [22, 35]. Previous transactional CC+ implementations provide either reading from a snapshot [18, 45, 47, 60, 61] or atomicity of updates [24, 61]. In Cure, a transactions provides both.

Taken together, the above features provide clear and strong semantics to developers. In fact, as Cure combines the three, it has the strongest semantics ever provided by an always-available data store.

Cure's design is based on a novel approach to support parallelism between servers within the data centre that minimises the overhead of causal consistency in inter-DC traffic [47]. Instead of the usual approach of checking whether a received update satisfies the causality conditions, which requires to wait for a response from a remote server —called explicit dependency check messages— Cure relies on dependency stabilisation, which makes updates visible in batches that are known to be safe according to causal consistency. Cure improves on previous work by encoding causal-order metadata as a single scalar per DC —thus incurring small overhead— to improve freshness and resilience to network partitions with respect to the state-of-art implementation of such mechanism (See Section 6.5).

To summarise, the contributions of this part of the thesis are the following:

- A novel programming model providing causally-consistent interactive transactions with high-level, conflict-free data types (Chapter 5.1).
- A high-performance protocol, supporting this programming model for geo-replicated data stores (Chapter 6).
- A comprehensive evaluation, comparing our approach to state-of-the-art data stores (Chapter 7).

This work is the result of collaboration with members of University of Kaiserslautern, Université Catholique de Louvain, and NOVA LINCS. It has been published in ICDCS'16 [16].

Chapter 5

Overview of Cure

5.1 Transactional Programming Model

A body of research has extended the causal+ consistency (CC+) model [60] by adding multi-key operations. There are two major efforts in this direction: *static read-only transactions* [18, 45, 47, 60, 61] that provide clients with a consistent view of multiple keys, and *update-only transactions* [61] that permit clients to perform atomic multi-key updates. Cure adds *general transactions* and *CRDT support* to ensure replica convergence.

General Transactions. Static-reads offer limited functionality. It is useful to perform reads in rounds, where the objects read in each round are selected according to the results of reads performed in the previous one(s). Consider the example of populating a news feed on a social network application with the most recent activity of a user's contacts. The application should first read the user's friends to retrieve their latest posts. Then, it should read the list of comments and likes of those posts, etc. Indeed, Facebook has reported that populating a user's news feed requires dozens of rounds alike [15]. With a static interface, the reads across rounds lose isolation guarantees, as they must be issued in different transactions. Our general transactions are interactive: they allow clients to combine read and write operations flexibly within the same transaction, when the read set is not known in advance. Under TCC, they ensure:

- Update *atomicity*, i.e., all updates occur and are made visible simultaneously.
- Transactions read from a snapshot that, as defined in Section 3.3.5, avoids causality violations and read skews. Moreover, Cure's transactions ensure all the session guarantees (Section 3.4).

Support for CRDTs. In Cure, an update is a CRDT-specific operation. For instance, a counter implements $increment(amount)$ and $decrement(amount)$, while a register's is $assign(value)$.

Cure ensures that updates are delivered to replicas exactly once and in causal order, which enables lightweight operation-based implementations of CRDTs (Section 3.5.2.3).

5.2 Programming interface

Cure's interactive interface offers the following operations:

- TxId ← START_TRANSACTION(CausalClock)
  initiates a transaction by creating a transaction coordinator process. This process returns a transaction handle that will be used when issuing reads and updates for that transaction. The system guarantees that reads in the transaction will observe updates no older than those encoded in the CausalClock vector clock. When CausalClock is not specified the server that receives the request creates a vector clock for the transaction that will include the most recent updates known to be available by this server.

  If START_TRANSACTION is not issued before the first read or update operation, the first operation starts the transaction implicitly, and additionally returns the transaction handle. In this case, it is also possible to pass a CausalClock as an argument. To simplify the explanation of the algorithms, we do not explain these cases in detail.

- Values ← READ_OBJECTS(Keys, TxId)
  returns the list of values that correspond to the state of the objects identified by the elements of the Keys list. The system guarantees that the values returned belong to a consistent snapshot.

- ok ← UPDATE_OBJECTS(Updates, TxId)
  declares a list of Updates for a transaction. Each update must respect the form:

  $(key, CRDT-type, operation, arguments)$, where key is the object identifier, $CRDT-type$ is the type of CRDT, $operation$ an operation exposed by $CRDT-type$, and $arguments$, the parameters the operation accepts. For instance, incrementing a counter identified by the key "my-counter" by two would look like: $update(my-counter, crdt-counter, increment, (2))$. The function returns ok.

- CommitTime ← COMMIT(TxId)
  commits the transaction identified by transaction handler TxId. It executes the updates (if any) and makes them visible to other transactions. It returns the transaction's CommitTime, which can be used by the client on further transactions (as a parameter of the START_TRANSACTION(CausalClock) operation). The process coordinating the transaction terminates.

- ok ← ABORT(TxId)
  discards the updates (if any) issued on behalf of the transaction and terminates the coordinator process.

  Static Interface. Cure also offers a static transactions. These receive a list of read and/or update operations. The system executes the transaction completely in a single call that starts, executes and commits the transaction. We do not illustrate this path to simplify explanation.

5.3 Design - causal consistency

Cure is designed with the goal of providing TCC in a cloud environment, while remaining highly-available under partition, without compromising scalability, and while serving fresh data. To meet these goals:

- Cure ensures that updates arriving from remote replicas are applied in causal order.
- Its design decouples propagating updates among replicas from making these updates visible respecting causal consistency. Partitions propagate updates pairwise, without requiring coordination with other partitions. A lightweight protocol involving all partitions runs asynchronously to establish the set of updates that are causally-consistent and thus safe to read.
- Causal order information is encoded using vector clocks sized with the number of replicas. Each partition relies on the timestamps taken from the physical clock to timestamp events, avoiding centrally assigned time-stamps [46].

In what follows: we explain how each of these design choices help us achieve our goals. In Section 16.1, we compare these design choices with those of other causally-consistent stores.

5.3.1 Updates applied in causal order for high availability.

In a geo-replicated setting, the requirement of availability under partition forces a replica receiving a remote update to verify that all the updates that causally precede (or causal dependencies) it were applied before making this update visible to readers. To illustrate the issue, consider an execution where update a to object A and b to object B, such that $a \rightsquigarrow b$, b arrives at a remote replica before a. A client reads B and sees the effects of b. Then, a network partition occurs such that a never arrives. To ensure causal consistency, the reader is not allowed to read A before a arrives, which would render this operation unavailable (or causally inconsistent).

5.3.2 Dependency stabilisation for scalability.

Ensuring that the causal dependencies of an update have been applied requires computation and, when the key space is partitioned, communication across servers (e.g., if the partitions storing

a and b are held by different servers). The traditional approach is *explicit dependency check messages*, where a partition receives a remote update and sends a message to each partition storing causal dependencies of the update. This mechanism is expensive in terms number of messages exchanged which affects throughput. It is possible to implementing an efficient broadcast protocol among partitions which results in negligible throughput degradation [47]. This is called *dependency stabilisation*, where partitions at a replica periodically exchange information regarding the updates they have received, and each partition uses this information to compute which updates have their dependencies satisfied. Cure follows this approach.

5.3.3 Vector clocks for serving fresh data.

Explicit dependency check messages normally rely on large causal-order metadata to reduce the number of per-update exchanged messages: by tracking dependencies more precisely, upon arrival of a remote update, a partition needs to contact fewer partitions. This results in small latencies to make a remote update visible (called update-visibility latency). On the contrary, GentleRain, which implements dependency stabilisation, relies on compact single-scalar timestamps [47]. To make an update with a given timestamp visible, a replica must wait to receive all updates with a smaller timestamp *from all other replicas*. This results in update-visibility latencies governed by the latency to the most distant replica and, under network partitions or failures, replicas do not apply updates that arrive from healthy remote replicas.

To improve visibility latency and progress without resorting to dependency check messages, Cure encodes dependencies in a vector clock sized with the number of replicas. This way, it makes the updates of a given replica visible independently of the state of others.

In the following chapter, we provide detailed protocol design, including and transaction execution, update propagation, and causal stabilisation.

Chapter 6

Protocol description

There are two kinds of processes involved in the execution of a transaction, a *transaction coordinator (TC)* which handles user requests from clients, and forwards it to *partitions (p)*, which store object versions, and reply to requests arriving from TCs to read and update the subset of objects they store. Every replica (or DC) follows an identical partitioning scheme. Note that partitioning is logical. A physical server can host multiple logical partitions.

Our protocol assumes that each server is equipped with a physical clock. Clocks are loosely synchronised by a time synchronisation protocol such as NTP [5]. Each clock generates monotonically increasing timestamps. The correctness of the protocol does not depend on the synchronisation precision. However, clock skew between servers can impact performance.

Cure annotates an update with a the commit time of its transaction, which is a vector timestamp with an entry per DC. Commit times produce a partial order that respects causal consistency. The protocol uses these commit times to make transactions visible in accordance with causality. Transactions originating at the local DC are immediately visible to clients when they commit, as their causal dependencies are automatically satisfied. In contrast, updates arriving from remote DCs depend on the *globally stable snapshot (*GSS*)*, which represents a consistent view of the store known to be available at all partitions in the local DC. A remote transaction is made visible when the GSS advances past their commit time. This ensures that all causally preceding transactions, i.e., that have a smaller commit timestamp, are already visible locally.

Cure keeps multiple versions of each object in order to read from a causally-consistent snapshot. Each version stores its value along with the vector timestamp that encodes its causal dependencies. Old versions are periodically garbage collected by the system.

6.1 Notation and definitions

Table 6.1 introduces the notation followed in this section for the state managed by transaction coordinators and storage partitions. We assume a total number of D DCs and P partitions. A partition m at DC d, denoted by p_d^m, keeps the following state:

cvc	Client causal vector timestamp
p_d^m	Partition m at DC d
$Clock_d^m$	Current physical time at p_d^m
pvc_d^m	vector timestamp at p_d^m
SS_d^m	Stable snapshot at p_d^m
$prepTx_d^m$	Prepared transactions at p_d^m
$committedTx_d^m$	Committed transactions at p_d^m
Log_d^m	Log of updates at p_d^m
PMC_d^m	Matrix of received pvc_d^i at p_d^m
T	Transaction
TC_T	Transaction Coordinator of T
svc_T	Snapshot vector timestamp of T
ct_T	Commit vector timestamp of T
$ws_T[m]$	Write set of T for partition m

Table 6.1: Notation used in the protocol description.

- pvc_d^m, a vector timestamp of size D, where position $pvc_d^m[k] = j$ indicates that p_d^m has received updates up to j from p_k^m, the partition that stores the same subset of the key space at DC k.

- SS_d^m, a vector timestamp of size D that denotes the latest consistent snapshot known by p_d^m to be applied by all partitions at DC. In order to advance SS_d^m, partitions of the same DC periodically exchange their pvc. Each p_d^m computes its SS_d^m as the aggregate minimum of all $pvc_d^i, \forall i \in 1 \ldots P$.

A client connects to a Cure server to issue a transaction. A server receiving a client request to start a transaction T starts a transaction coordinator process *(TC_T)*, which lives throughout the lifetime of the transaction.

6.2 Transaction Execution

Algorithms 6.1 and 6.2 show the pseudocode of the protocol for executing transaction T at DC d followed by the transaction coordinator (TC_T) and the partitions involved, respectively.

Start transaction. The transaction coordinator TC_T starts by ensuring that it assigns T a snapshot no older than the last one seen by the client, represented by cvc (Alg. 6.1, line 3). This is necessary to ensure that clients observe monotonically increasing causally consistent views of the data store.

To define the causally consistent snapshot that T will access, TC_T sets the vector timestamp svc_T to include all remote transactions that are stable in the local DC, plus all locally committed

Algorithm 6.1 Transaction coordinator at server m of DC d

```
1:  function START_TRANSACTION(cvc)
2:      for k = 1...D, k ≠ d do
3:          wait until cvc[k] ≤ SS_d^m[k]
4:      allocate T
5:      svc_T ← SS_d^m
6:      svc_T[d] ← MAX(cvc[d], Clock_d^m)
7:      return T
8:
9:  function UPDATE_OBJECTS(T, Updates)
10:     for all ⟨Key, Operation⟩ ∈ Updates do
11:         p_d^i ← PARTITION(Key)
12:         if p_d^i ∉ UpdatedPartitions_T then
13:             UpdatedPartitions_T ← UpdatedPartitions_T ∪ {p_d^i}
14:         ws_T[i] ← ws_T[i] ∪ {⟨Key, Operation⟩}
15:     return ok
16:
17: function READ_OBJECTS(T, Keys)
18:     for all Key ∈ Keys do
19:         p_d^i ← partition(Key)
20:         Val ← send READ_KEY(svc_T, Key) to p_d^i
21:         for all ⟨Key, Operation⟩ ∈ ws_T[i] do
22:             Val ← APPLY_OPERATION(Val, Operation)
23:         Values ← Values ∪ {Val}
24:     return Values
25:
26: function COMMIT(T)
27:     if UpdatedPartitions_T = ∅ then
28:         return svc_T
29:     for all p_d^i ∈ UpdatedPartitions_T do
30:         send PREPARE(T, ws_T[i], svc_T) to p_d^i
31:         wait until received (T, PrepTime) from p_d^i
32:     CommitTime ←MAX(all prepare times)
33:     ct_T ← svc_T
34:     ct_T[d] ← CommitTime
35:     for all p_d^i ∈ UpdatedPartitions_T do
36:         send COMMIT(T, ct_T) to p_d^i
37:     return ct_T
```

Algorithm 6.2 Protocol executed by partition p_d^m

```
1:  function READ_KEY(svc_T, Key)
2:      wait until svc_T[d] ≤ pvc_d^m[d]
3:      Val ← SNAPSHOT(Key, svc_T, Log_d^m)
4:      send Val to TC_T
5:
6:  function PREPARE(T, ws_T[m], svc_T)
7:      wait until svc_T[d] ≤ Clock_d^m
8:      PrepTime ← Clock_d^m
9:      Log_d^m ← Log_d^m ∪ {⟨ws_T[m], PrepTime, svc_T⟩}
10:     prepTx_d^m ← prepTx_d^m ∪ {⟨T, PrepTime⟩}
11:     send ⟨T, PrepTime⟩ to TC_T
12:
13: function COMMIT(T, ct_T)
14:     Log_d^m ← Log_d^m ∪ {⟨ct_T⟩}
15:     prepTx_d^m ← prepTx_d^m \ {⟨T, PrepTime⟩}
16:     committedTx_d^m ← committedTx_d^m ∪ {⟨T, ct_T⟩}
17:
18: function PROPAGATE_TXS()                                  ▷ Run periodically
19:     if prepTx_d^m ≠ ∅ then
20:         pvc_d^m[d] ← MIN(prepTx_d^m) − 1
21:     else
22:         pvc_d^m[d] ← Clock_d^m
23:     if committedTx_d^m = ∅ then
24:         for k = 1...D, k ≠ d do
25:             send HEARTBEAT(pvc_d^m[d], d) to p_k^m
26:         return
27:     for all ⟨T, ct_T⟩ ∈ committedTx_d^m | ct_T < pvc_d^m[d] do
28:         for k = 1...D, k ≠ d do
29:             send REPLICATE_TX(ws_T[p], ct_T, svc_T, d) to p_k^m
30:             committedTx_d^m ← committedTx_d^m \ {⟨T, ct_T⟩}
31:
32: function REPLICATE_TX(ws_T[p], ct_T, svc_T, k)
33:     Log_d^m ← Log_d^m ∪ {⟨ws_T[p], ct_T, svc_T⟩}
34:     pvc_d^m[k] ← ct_T[k]
35:
36: function HEARTBEAT(TimeStamp, k)
37:     pvc_d^m[k] ← TimeStamp
38:
39: function BCAST_PVC()                                      ▷ Run periodically
40:     for i = 1...N, i ≠ m do
41:         send UPDATE_GSS(m, pvc_d^m) to p_d^i
42:
43: function UPDATE_GSS(i, pvc)
44:     PMC_d^m[i] ← pvc
45:     for k = 1...D, k ≠ d do
46:         SS_d^m[k] ← min_{i=1...N} PMC_d^m[i][k]
```

transactions. The former is achieved by setting the vector to the value of SS_d^m (Alg. 6.1, Line 5), while the latter is achieved by setting the entry for the local DC in svc_T to the maximum of either the physical clock of the server or the client's previously observed timestamp (Alg. 6.1, Line 6). The transaction's snapshot includes the updates of all transactions that have a commit vector timestamp smaller than or equal to svc_T. This guarantees that the snapshot is causally consistent, since it includes the dependencies of all transactions.

Update objects. To update, a client provides a list of key-update pairs, which TC_T buffers in a per-partition write set ($ws_T[m]$) to be sent, when committing T, to each updated partition at DC d, and replies with an *ok* response.

Read objects. To read, the client provides a list of keys. TC_T forwards a read request to each local partition (retrieved by the call to the PARTITION function) storing some desired objects. Upon receiving such request, and in order to ensure that the snapshot includes all updates with commit time smaller than svc_T, p_d^m might need to wait for its $pvc_d^m[d]$ to catch up (Alg. 6.2, Line 2). Once this is satisfied, p_d^m returns the latest version of the object with commit time no newer than svc_T, which is retrieved by calling the SNAPSHOT function (Alg. 6.2, Line 3). When TC_T receives this reply, it applies the update operations on the same object (if any) issued by T during previous UPDATE_OBJECTS operations (Alg. 6.1, Line 22), generating a new version of the object. Note that this is a consequence of providing more developer-friendly data types than just basic registers. TC_T caches this result until all objects in the operation are read. This process is repeated for every requested key. Once it finishes, TC_T returns all read values to the client.

Commit. When receiving a commit request from a client, TC_T starts a two-phase commit (2PC) protocol to atomically commit the updates of transaction T at local DC d. In the first phase, TC_T sends a prepare message including $ws_T[m]$ to each of the updated partitions (Alg. 6.1, Lines 29-31). Upon receiving such message, each partition takes the current value of its physical clock (Alg. 6.2, Line 8) and proposes it as the transaction's commit timestamp. Next, it stores its write set in its log. We use the abstraction of a log to illustrate persistent storage. Internally, the system stores an in-memory cache of CRDT object versions and operations where reads are served from. TC_T computes the transaction's commit timestamp as the maximum of all proposed prepare timestamps (Alg. 6.1, Line 32), and generates ct_T, the commit vector timestamp of T, by applying this commit time, at position d, to svc_T. Following, the coordinator sends a commit message that includes the transaction's commit vector timestamp. to all involved partitions

When a partition receives the commit message, it removes T from $prepTx_d^m$, stores the ct_T in its log (this can be done asynchronously depending on the recovery mechanism in place), and adds T and its commit timestamp to $committedTx_d^m$ for propagating its updates to the other DCs.

Choosing the maximum of the proposed timestamps as the commit timestamp of a transaction is important for correctness. The read protocol waits for prepared transactions expected to be included in a snapshot (Alg. 6.2, Line 2). If TC_T were to choose a ct smaller than the prepare

timestamp of some participant partition, a transaction reading from the partition with svc smaller than the prepare timestamp but greater than this ct would not be delayed to include the committing transaction. Therefore, it would read from an inconsistent snapshot.

6.3 Replication and stable snapshot computation

Each partition periodically synchronises with its sibling partition in other DCs. When there are no new updates to send, a heartbeat is sent to indicate remote partitions that the partition's clock has advanced. This allows the remote replica to make updates visible arriving from other partitions. Upon receiving a heartbeat (Alg. 6.2, Line 36), a replica advances $pvc_d^m[k]$, thus acknowledging that it has received all updates from DC k, up to the received timestamp. When there are updates to send, a replica sends, in commit-time order, all committed updates with timestamp smaller than any prepared but not yet committed transaction (Alg. 6.2, Lines 27-29).[1] On receiving an update replication message from DC k, a replica inserts the received updates in its log and advances $pvc_d^m[k]$, setting it to the update's commit timestamp $ct_T[k]$.

Our algorithm decouples propagating updates among replicas from making these updates visible. An update received from a remote replica is only made visible after it is known that all updates from the same transaction (and their dependencies) have already been received at all partitions. To this end, partitions in each DC exchange their pvc_d vectors in the background (Alg. 6.2, Line 39), and each partition m computes its SS_d^m as the aggregate minimum of known pvc_d (Alg. 6.2, Line 43).

6.4 Correctness

We provide an informal proof that Cure implements TCC by showing that the snapshot read by a transaction is causally consistent, and respects the atomicity of committed transactions.

Proposition 1. Version commit vectors respect causal order. If an update u_1 depends on an update u_2, then $u_2.ct < u_1.ct$.

An update u_1 depends on u_2 if the transaction of u_2 reads from a snapshot that contains u_1. From Alg. 6.2 line 7, a proposed timestamps is always greater than its snapshot time (in DC d, the entry d of its snapshot vector timestamp). Since the commit timestamp is generated as the maximum of proposed timestamps, the commit time of a transaction is always greater than its snapshot time. Then, by Alg 6.1 lines 33-34, the commit vector timestamp of an update is always greater than its snapshot vector timestamp.

Proposition 2. A partition vector timestamp $pvc_d^m = t$ implies that p_d^m has received all updates with commit vector timestamp $\leq t$.

[1] A transaction being prepared with a given prepare timestamp can commit before a concurrent transaction with a lower one when they update different partitions.

First, we show that the proposition is valid for remote updates. We prove this by contradiction. Assume there is a remote update u from DC j such that $u.ct < t$, and p_d^m has not received u. By Alg. 6.2 lines 33-34, the partition would have received an update u_1 such that $u_1.ct[j] = t[j]$. Because the updates are sent in the order of their timestamps, the partition cannot receive another update u_1 before u if $u_1.ct[j] > u.ct[j]$. Hence $u.ct[j] > t[j]$, implying $u.ct \not< t$, leading to the contradiction.

Now we show that there are no pending local updates with commit vector timestamp $\leq t$. When updating $pvc[d]$, the partition finds the minimum prepared time stamps of the transactions in the prepared phase. Since the physical clock is monotonic and the commit time is calculated as the maximum of all prepared times, it is guaranteed that all future transactions will receive a commit time which is greater than or equal to this minimum prepared time stamp. So, when the $pvc[d]$ is set to the minimum prepared time minus 1 (Alg. 6.2, Line 20), the partition has already received all updates for the snapshot pvc.

Proposition 3. Reads return values from a causally consistent snapshot.

When a transaction attempts to read from a given snapshot, it waits until pvc includes the snapshot time (Alg 6.2 line 2). This ensures the partition will not further commit any transaction with commit vector smaller than the transaction's, as the local position of pvc advances to time t when there are no further transactions which have proposed a prepare time smaller or equal than t (Alg 6.2 lines 20-22). By Proposition 2, all updates from remote sibling partitions with commit vector smaller or equal to the transaction's snapshot time have been applied locally, in causal order. Therefore, the read returns values from a causally consistent snapshot. As this occurs at every partition, reading from many partitions also ensures reads are causally consistent.

Proposition 4. Reading from a snapshot respects atomicity.

Atomicity is not violated even though updates (local and remote) are made visible independently by each partition. All updates from a transaction belong to the same snapshot because they receive the same commit vector timestamp. The proof of this proposition follows directly from Proposition 3. A read is delayed until the same snapshot is available at all (accessed) partitions, thus reading all or no updates from a transaction.

From Propositions 1-4, it follows that Cure implements TCC, since every transaction reads from a causally consistent snapshot that includes all effects of its causally-preceding transactions.

6.5 Discussion

6.5.1 Session Guarantees

Cure ensures that the transactions of a client see (i) the effects of previously committed transactions by the same client, and (ii) monotonically-non-decreasing snapshots of the data store.

When a client finishes a read-only transaction, its snapshot vector timestamp is returned. Similarly, when a client successfully commits an update transaction, its commit vector timestamp is returned. A client must keep this vector timestamp, called cvc. When a client starts a new transaction, it sends cvc with its request. In the unlikely case where cvc is greater than the SS at the server receiving the request, the client is blocked until SS proceeds past cvc. Otherwise, it starts immediately.

6.5.2 Efficient SS computation

Under Cure, partitions within the same DC periodically exchange their pvc to compute their SS. To do this efficiently, Cure builds a tree over all servers in a DC and computes an aggregate minimum using the tree [47]. When compared to a simple broadcast approach, this reduces the number of messages exchanged in the network, while computing and distributing SS in a reasonable amount of time. This is important for remote update visibility latency as the updates of a remote transaction are only made visible after SS passes the transaction's commit vector timestamp.

6.5.3 Garbage Collection

Each partition periodically garbage-collects object versions that will no longer be accessed by any transaction (not depicted in the algorithm). Using the same broadcast mechanism as update stabilisation, a partition periodically sends to all other partitions at its site the minimum snapshot vector timestamp of its active transactions. Upon collecting this information from all partitions, a partition computes the aggregate minimum. This computed vector is then used to remove versions older than the version with higher timestamp that is smaller or equal to the computed minimum, which are guaranteed to be never accessed again.

6.5.4 Support for CRDTs

Cure offers support for operation-based CRDTs (Section 3.5.2.3). Their implementation requires adequate support from the system, as an object's value is defined not just by the last update, but also by the state it is applied on. This requires that updates are applied exactly once, and in causal order. Cure encodes, with each update, the snapshot vector timestamp of its transaction. This vector represents the state over which the update must be applied at a remote DC. To ensure that an update is applied exactly once, partitions assign, to each update-propagation message, a totally ordered unique identifier. In the absence of new updates to send, a partition sends a heartbeat including a unique identifier. A partition receiving a transaction or heartbeat from a sibling partition uses this timestamp to detect missing and duplicate messages.

Chapter 7

Evaluation of Cure

7.1 Setup

We built Cure as part of Antidote [4], an open-source cloud database. Antidote is built using Erlang/OTP, a functional language designed for concurrency and distribution. To partition the set of keys across distributed, physical servers we use riak_core [82], an open source distribution platform using a ring-like distributed hash table (DHT), partitioning keys using consistent hashing. Key-value pairs are stored in an in-memory hash table, with updates being persisted to an on disk operation log using Erlang's disk-log module [48].

For comparison, we implemented a protocol that ensures eventual consistency and Read Committed isolation. This protocol single versioned. It supports LWW registers, where the ordering of concurrent updates is determined by physical clocks. It also supports an Erlang library of state-based CRDTs called riak_dt [83]. We also implemented two state-of-art causally-consistent protocols, Eiger and GentleRain. Eiger implements causal consistency and supports LWW registers. It tracks one-hop nearest dependencies, and uses explicit dependency checks to apply updates in causal order. GentleRain uses a dependency stabilisation mechanism similar to Cure's, but encodes causal-order information in a single scalar. In addition to LWW registers, Cure supports operation-based CRDTs. Cure, Eiger and GentleRain guarantee consistent, (static) read-only and atomic update transactions. Cure additionally supports interactive read and update transactions. Objects in Cure, Eiger, and GentleRain are multi-versioned. For each key, a linked list of recent updates and snapshots is stored in memory. An update operation appends a new version of the object to the in-memory list and asynchronously writes a record to the operation log. Old versions are garbage-collected following the mechanism described in Section 6.5.

The following experiments are run using a variable number of DCs, each comprised of a variable number of servers. Nodes within the same DC communicate using the distributed message-passing framework of Erlang/OTP running over TCP. Connections across separate DCs use ZeroMQ [6] sockets running TCP. Each server connects to all other servers to avoid any centralisation bottlenecks. To simulate the DCs being geo-distributed, we added a 50ms delay to

all messages sent over ZeroMQ. Lost messages are detected at the application level and resent.

Hardware. All experiments are run on the Grid5000 [51] experimental platform using dedicated servers. Servers were located located within a cluster in Rennes. Each server consists of two Intel Xeon E5-2630 v3 CPUs, with eight cores/CPU, 126GB RAM, and two 558GB hard drives. Nodes are connected through shared 10Gbps switches. The measured latency among servers in the cluster, over TCP/IP, was approximately 0.15 ms. Before running each experiment, clocks were synchronised using an NTP [5] server running within the cluster.

Workload generation. The data set used in the experiments includes 100k key-value pairs per server with each pair being replicated at all three DCs. Tests are performed with LWW registers and CRDT sets.

We use a custom version of Basho Bench [1] to generate load. A client repeatedly runs single-operation transactions of either a read or an update. To select a key, a client uses a power-law distribution. The ratio of reads to updates varies depending on the benchmark. For Cure, Eiger, and GentleRain, transactions ensure the Read-Your-Writes session guarantee. When committing, a transaction returns a commit timestamp. The client passes this timestamp as an argument when issuing a subsequent transaction. Each protocol uses this information to ensure a transaction observes state no older than that encoded in the timestamp. Clients run on their own physical machines, with a ratio of one client server per five Cure servers. We found this ratio to sufficiently load the system, without over-stressing it for any workload. Each client server uses 40 threads to send requests at full load. Each instance of the benchmark is run for two minutes, the first minute being a warm-up period. Google's Protocol Buffer interface is used to serialise messages between Basho Bench clients and Antidote servers.

7.2 Cure's scalability

To evaluate the scalability of Cure, we run a single-DC configuration and vary the number of servers from 5 to 25. We run the same experiment on 2 and 3 DCs comprised of 25 servers each (50 and 75 Cure servers in total, respectively). In both cases the read/update ratio varies from 99/1 percent read/write to 50/50 percent read/update ratio. Objects are LWW registers an an update assigns random binary values of 1 KB.

As Figure 7.1 shows, throughput increases 4.8 times when going from 5 to 25 nodes within a single DC, under all workloads. Furthermore, on the configurations of 2 and 3 DCs consisting of 25 servers each, we observe a 1.8x and a 2.8x respective increase for 99 percent reads, and 1.8x and 2.6x for 50 percent writes when compared to a single DC with the same number of servers.

The observed scalability is expected due to the decentralised design of Cure. Still, numbers do not show a perfect linear progression due to the cost of replicating updates across DCs and because the background stable time calculation becomes more expensive as the number of servers increases per DC.

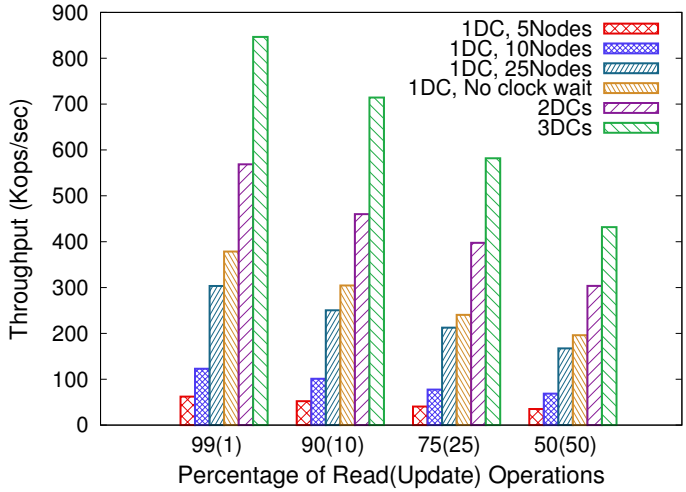


Figure 7.1: Scalability of Cure

Latency. For this experiment, the median latency for reads was 0.7 ms for all workloads. The median for updates varied from 1 to 2ms when increasing the update rate. Writes are more expensive than reads, as they require updating in memory data structures and writing to disk. Additionally, given that updates are replicated 3 times, they create a greater load on the system than reads.

Impact of waiting. In order to evaluate the impact of clock skew in performance, we implemented an unsafe version of Cure that avoids waiting for a snapshot to be ready at a partition receiving a read request (Alg. 6.2, Line 2). The No clock wait bar shows the throughput obtained by this protocol when run at a single DC consisting of 25 servers, which displays up to 1.25x increase when compared to the correct implementation, under the read dominant workload.

7.3 Comparison to other systems

To evaluate the performance of Cure when compared to other protocols, we run a three-DC benchmark with 25 servers per DC, varying the update to read ratio. We compare all systems using LWW registers, and Cure to eventual consistency using CRDT sets. Figures 7.2 and 7.3 show the results.

LWW registers. We compare all systems using LWW registers of 1 KB values each (Figure 7.2). Unsurprisingly, eventual consistency performs better than all other protocols, outperforming Cure by approximately 30 percent across all workloads. Under eventual consistency, reads and updates are cheaper, since they are single versioned and do not require processing causal dependencies.

Under the 99/1% read/write workload, causally-consistent systems perform similarly to each other. At this read-write ratio, the amount of dependency checks performed by Eiger is small. As

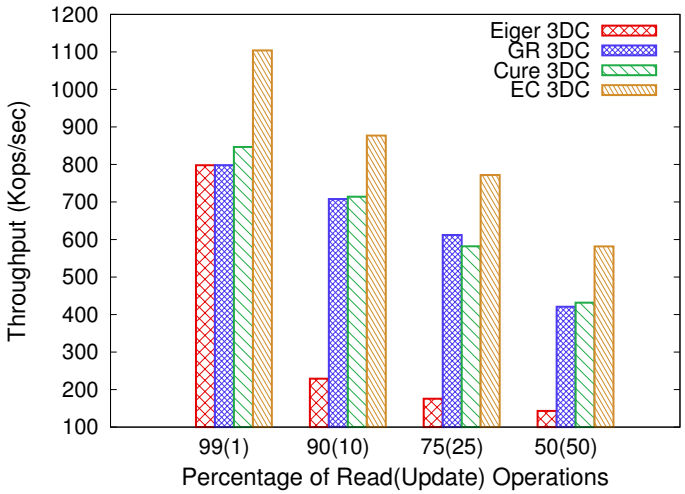


Figure 7.2: Comparison of Cure to other systems using LWW registers.

soon as the update rate is increased to 10 percent, the cost of explicitly checking dependencies increases dramatically and the throughput of Eiger degrades. This trend continues throughout higher-update-rate workloads. When compared to Cure, we observe a small additional overhead for GentleRain, which normally needs to retrieve slightly older versions of objects than Cure. This happens due to its larger remote update visibility latency, which incurs extra processing of lists of object versions.

CRDT sets. We compare Cure to the eventually-consistent protocol using CRDT sets (Figure 7.3). Cure supports operation-based CRDTs, where objects transfer updates among replicas. On the contrary, the eventually-consistent protocol requires state-based CRDTs (as explained in Chapter 3.5.2.3, operation-based CRDT require causal delivery of operations), where replicas exchange object state.

For this experiment, we use "small" and "big" sets that grow up to 10 and 100 elements of 100 bytes each (1 and 10 KB in total), respectively. Once sets reach this size, the workload balances the amount of add and remove operations to keep their average size constant.

For both sizes of sets, we observe a similar behaviour. As observed in the LWW-register experiment, under the 99/1% read/write workload, eventual consistency outperforms Cure. For 90/10% reads/writes, this difference becomes smaller. Finally, at higher update rates, Cure overtakes eventual consistency's performance. The eventually-consistent protocol transfers and processes CRDT state (1 and 10 KB for small and big sets respectively). Under Cure replicas transfer operations (100 bytes to perform an add operation).

Update visibility latency. To calculate the stable time, each node within a DC broadcasts its vector clock to other nodes within the DC at a frequency of 10ms. Additionally, heartbeats between DCs are sent at a rate of 10ms in the absence of updates.

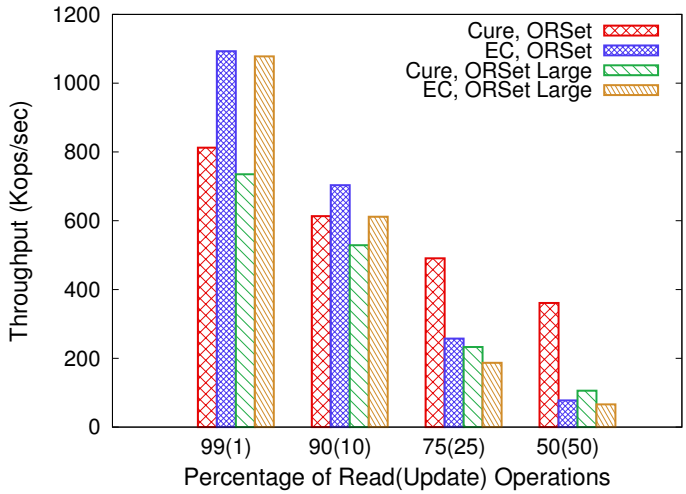


Figure 7.3: Comparison of Cure to an eventually-consistent system using CRDT sets.

For all experiments, we measured the remote visibility latency observed by DC 1 for updates coming from DCs 2 and 3. Under Cure, we observed an average remote update visibility latency of between 80 and 90 ms for updates originating at DCs 2 and 3. Under GentleRain, we observed a visibility latency of 90 ms for both DCs 2 and 3. Moreover, under the update-intensive workloads, we observed frequent short lived peaks of around 150 ms visibility latency for one or both of the DCs due to the cost of processing external updates. Under such conditions, Cure only observed that delay for updates from the affected DC while under GentleRain, the visibility latency of both DCs was penalised under load. The use of a single scalar penalises GentleRain, which is able to make updates visible at the rate of the slowest DC (See Section 5.3). By using a vector clock, Cure is able to make updates coming from different DCs visible independently (As explained in Section 5.3.3).

Chapter 8

Conclusion of Part I

In this part of the thesis, we have introduced Cure, a transactional protocol for distributed and geo-replicated storage. Cure offers the strongest properties achieved so far that remain highly available: Transactional Causal Consistency with CRDT support through an interactive transactional interface. Its design is available under network failures. It offers high throughput, and penalises minimally the latency required to make an update visible. We have evaluated Cure's implementation, showing that its scalability is similar to eventual consistency, while offering stronger semantics. Our comparison to existing causally-consistent systems shows that Cure exhibits better performance, smaller update visibility latency, and better progress.

Part II

The three-way trade-off: Read Isolation, Latency and Freshness

Chapter 9

Introduction to Part II

In this part of the thesis, we study the costs of reading data in a distributed, transactional storage system. In particular, we try to understand whether it is possible to provide strong read guarantees while ensuring both fast response and fresh data. It is well known that stronger guarantees will come with higher costs: protocols rely on blocking, retrying operations, or reading from the past to isolate transactions. Other systems completely eschew isolation to avoid these costs: a recent paper from Facebook (whose performance is strongly read-dominated) states:*"stronger properties have the potential to improve user experience and simplify application-level programming [... but] are provided through added communication and heavier-weight state management mechanisms, increasing latency [...] This may lead to a worse user experience, potentially resulting in a net detriment"* [15]. Is this wariness justified, i.e., is it inherently impossible to combine fast reads with strong guarantees, or can the situation be improved by better engineering? This work provides a formal and operational study of the costs and trade-offs. We formalise the three-way tension between read guarantees, read delay (and hence latency), and freshness, and show the desirable points of the design space which are possible/impossible.

Because non-serialisable guarantees can improve performance and availability, in this work, we do not necessarily assume that updates are totally ordered.[1] Furthermore, we allow weakening read isolation: in addition to Atomic Visibility, the strongest guarantee, which is assumed by classical transactional models, we identify (the weaker) Order-Preserving Visibility, which ensures the absence of ordering anomalies but allows Read Skews (Section 3.3.3) and consider (the weakest) Committed Visibility, which ensures read guarantees equivalent to Read Committed isolation (Section 3.3.5.3). Finally, we also consider the *freshness* dimension, because (as we show) decreasing read delay sometimes forces to read a version of the data that is not the most recent.

Figure A.1 illustrates the three-way trade-off between the guarantees, delay, and freshness of transactional reads. For instance, under Order-Preserving and Atomic Visibility, it is possible to read with no extra delay (compared to a non-transactional system), but then the freshest data is

[1] In replicated systems, enforcing a monotonic total order of updates enables Strong Consistency under Partition (CP); but, conversely, Availability under Partition (AP) requires accepting concurrent updates [79].

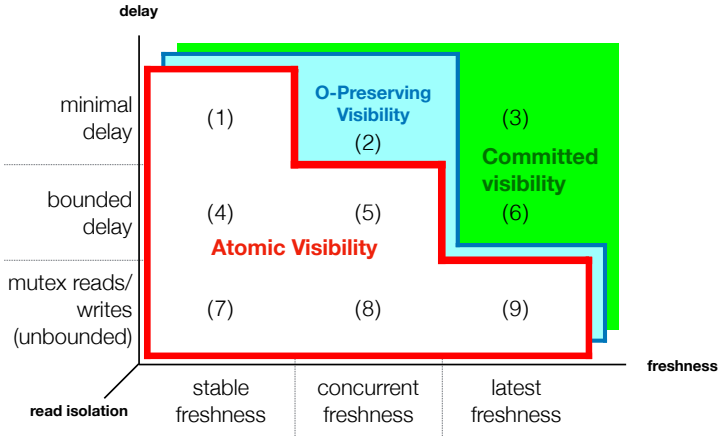


Figure 9.1: The three-way trade-off. The boxed areas represent possible guarantee/read delay/ freshness combinations. Upwards and right is better performance; guarantees get stronger from the back to the front planes. Combinations missing from the picture are impossible.

not accessible. Transactions that access the freshest data with no extra delay are only possible under Committed Visibility (Sector 3). However, we show that minimal-delay Order-Preserving reads allow observing updates of concurrently-committed transactions (Sector 2). As we will see in our evaluation chapter, this allows for a significant freshness improvement over Atomic Visibility, which forces reading data that was stable (written and acknowledged) before the transaction started (Sector 1). If, on the other hand, the application requires the freshest data, under either Atomic or Order-Preserving Visibility, this is possible only under a protocol where reads and writes are mutually exclusive, e.g., a read might be delayed (blocked, or in a retry loop) indefinitely by writes, or vice-versa (Sector 9).

This work includes the following contributions:

1. A formal study of the trade-offs between the read guarantees, delay, and freshness of transactional reads. We prove which desirable combinations are possible and which are not.
2. Two new isolation properties, TCC$^-$ and PSI$^-$, which result from degrading the read guarantees of TCC and PSI, from Atomic Visibility to (causal) Order-Preserving, which positions these models differently with respect to the trade-off.
3. Three minimal-delay protocols guided by the results of our analysis: AV ensuring TCC at Sector 1, OP ensuring TCC$^-$ at Sector 2, and CV ensuring Read Committed Isolation at Sector 3. We provide detailed protocol design, including pseudo-code. To our knowledge, these protocols are the first to offer these guarantees in a latency-optimal fashion.
4. An evaluation of these protocols to empirically validate our theoretical results. In our measurements, we compare the introduced protocols to Cure, which ensures Atomic Visibility, concurrent freshness, and bounded delays because of blocking scenarios (Sector 5). Our minimal-delay protocols exhibit similar latency. CV always observes the most recent data, whereas freshness degrades negligibly for OP, and severely under AV.

We expect these results help system designers make decisions when selecting and building transactional storage.

Earlier versions of this work were published in PaPoC'15 [86], SRDSW'15 [88], and PaPoC'16 [87].

Chapter 10

Requirements

10.1 Transactions

The application consists of *transactions*. A transaction consists of any number of reads, followed by any number of writes, and terminated by an abort (writes have no effect) or a commit (writes modify the store; in what follows, we consider only committed transactions). The transaction groups together low-level storage operations into a higher-level abstraction, with properties that help developers reason about application behaviour.[1] These are often summarised as the ACID properties (Section 3). Atomicity and durability will be taken for granted in the trade-off analysis. Interestingly, our results are independent of the write model, e.g., totally ordered or not. Therefore, we do not assume strong consistency or isolation (e.g., serialisability). Specifically, neither writes nor reads are necessarily totally ordered, and we consider several read guarantees.

For simplicity, we assume that a transaction reads a data item at most once (and similarly for writes). The set of item states read by a transaction is called its *snapshot*. Our study distinguishes some important properties of a snapshot, explained in the next few sections: snapshot guarantees, delay, and freshness.

10.2 Snapshot guarantees

Snapshot guarantees constrain the states of the data items that can be accessed by a given snapshot. The stronger guarantees provide higher isolation, and thus facilitate reasoning by the application developer. As we shall see, the weaker ones enable better performance along the freshness and delay metrics.

We distinguish three levels of snapshot guarantees, which will be defined formally later (in Section 11.1): Committed, Order-Preserving, and Atomic Visibility. Table 10.1 summarises the guarantees of each level in terms of read-related anomalies.

[1] A model that does not support transactions is identical to one where each individual read or write operation is wrapped in a transaction that commits immediately.

Read Anomalies* / *Snapshot	**Committed**	**Order Preserving**	**Atomic**
Dirty Read	x	x	x
Order (e.g. causal) violation	-	x	x
Read Skew	-	-	x

Table 10.1: Snapshot guarantees - Anomaly comparison

10.2.1 Committed Visibility

At the weakest level, **Committed Visibility**, a snapshot may include any updates that have been committed. As it sets no constraints between items, it allows many read anomalies. Committed Visibility offers read guarantees equivalent to those of Read Committed Isolation (RC).

10.2.2 Order-Preserving Visibility

We identify **Order-Preserving Visibility**, an intermediate level that strengthens Committed Visibility by ensuring that the snapshot preserves a (partial or total) order relation O. O might be the (partial) happens-before order, in order to enforce causal consistency [14, 56], or the total order of updates in the context of a strong isolation criterion such as Serialisability or Snapshot Isolation [27, 29].

Specifically, Order-Preserving Visibility ensures that transactions do not observe gaps in a prescribed order relation. Consider, in a social network, the data items *photos* and *acl* representing user Alice's photo album and the associated permissions. The set of their states (initially $photos_0$ and acl_0) is ordered, for instance, by causal order (defined in Section 1). Alice changes the permissions of her photo album from public to private (new state acl_1), then adds private photos to the album (state $photos_2$). Thus, $acl_0 \rightsquigarrow acl_1 \rightsquigarrow photos_2$. Unlike Committed Visibility, Order-Preserving Visibility disallows the situation where Bob would observe the old permissions (acl_0) along with the new photos ($photos_2$), missing the causally-related update, i.e., observing a gap, that restricted the permissions (acl_1). This pattern, where the application enforces a relation between two data items by issuing updates in a particular order, is typical of security invariants [79]. It also helps to preserve referential integrity (create an object before referring to it, and destroy references before deleting the referenced object). Under causal order, order-preserving reads have been called causally-consistent snapshot reads [14].

In Chapter 12, we introduce new isolation levels that result from combining Order-Preserving reads with different update/commit guarantees.

10.2.3 Atomic Visibility

Atomic Visibility is the strongest read isolation level. It is order-preserving, and additionally disallows read skews (Section 3.3.3): if the transaction reads some data item written by another

transaction, then it must observe all updates written by that transaction (unless overwritten by a later transaction). Atomic Visibility is provided by all models that disallow the read skew anomaly (see Table 3.1).

As we saw in Chapter 3.2, Atomic updates serve to maintain equivalence or complementarity between data items [79]; for instance, ensuring in a bank application that during a transfer Alice's account is debited a certain amount if and only if this amount is credited to Bob's. Atomic Visibility ensures that a transaction will observe both updates, or none, thus forbidding other transactions from observing a state where Bob's account has been credited, and Alice's not debited (or vice-versa).

10.3 Delay

Serving requests with low latency keeps users engaged and directly affects revenue (Section 2.2). Read latency is also an important performance metric for services that are heavily read-dominated, such as social networks. For instance, serving a Facebook page requires tens of rounds to read thousands of items for a single page [33]. Each round reads many items. A round influences what is read in the next round.

10.3.1 Minimal Delay

The fastest read protocol is one that addresses multiple servers in parallel within a round, and where any one server responds immediately, in a single round-trip, without coordinating with other servers. Intuitively, this design makes it difficult to ensure strong snapshot guarantees.

We will characterise protocols by estimating the *added delay* above this baseline, called *Minimal delay*. Systems that ensure minimal delays include Linkedin's Espresso [73] and Facebook's Tao [33], which offer no isolation.

10.3.2 Bounded delay

A protocol exhibits bounded delay when it requires sequential reads (i.e., parallel reads are not supported), a bounded number of retry round-trips may occur to read from a server, and/or a server may block for a bounded amount of time before replying to a read request. For instance, in Cure a server might block for a bounded amount of time to wait for clocks to catch up, or for transactions to commit. Thus, Cure exhibits bounded delay.

10.3.3 Mutex reads/writes (or unbounded delay)

A protocol exhibits unbounded delays —or, equivalently, requires mutually-exclusive reads and writes— when a read might be delayed indefinitely by writes, or vice-versa, because the protocol

disallows the same data item from being read and written concurrently (e.g., Google's Spanner strictly-serialisable transactions [40]).

10.4 Freshness

Another important metric is how recent is the data returned by a read. Users prefer recent data [3]. Some isolation levels (e.g., Strict Serialisability) require data to be the latest version. Under others (e.g., Snapshot Isolation), serving recent data makes aborts less likely and hence improves overall throughput [71, 75]. Storing only the most recent version of a data item enables update-in-place and avoids the operational costs of managing multiple versions.

However, MVCC protocols [28] maintain multiple versions of a data item to ensure snapshot guarantees. Serving an old item may be faster than waiting for the newest one to become available; indeed, it would be easy for reads to be both fast and isolated, by always returning the initial state.

Freshness is a qualitative measure of whether snapshots include recent updates or not. We consider three degrees of freshness, latest, stable, and concurrent.

10.4.1 Latest Freshness

The most aggressive is **Latest Freshness**, which guarantees a server always returns the most-recent committed version of any data item that it stores, at the moment which it replies to a read request. Because they do not make snapshot guarantees, intuitively, systems like Espresso and Tao [33, 73] can read with minimal delay under latest freshness.

10.4.2 Stable Freshness

The most conservative is **Stable Freshness**, which enables fast reads by returning data from a *stable snapshot*, i.e., one known to be ready when the transaction started. Therefore, stable freshness prevents a transaction from reading the updates of other transactions that concurrently update the objects it reads. Spanner's serialisable read-only transactions [40] exhibit stable freshness. When a read-only transaction starts, it is assigned a timestamp t that guarantees that no transaction running in the system will commit with timestamp $\leq t$.

10.4.3 Concurrent Freshness

Finally, the intermediate **Concurrent Freshness** does not necessarily return the latest version. It allows a server to read updates that are not stable. For instance, it allows reading the updates of a committed transaction that ran concurrently with the reader. For instance, COPSs exhibits concurrent freshness [60].

10.5 Optimal reads

We say a protocol has optimal reads if it ensures both minimal delay and latest freshness. An optimal-read protocol is one that supports parallel reads, and where a server is always able to reply to a read request immediately, in a single round trip, with the latest committed version that it stores.

Chapter 11

The three-way trade-off

In this section, we study the three-way trade-off between transactional reads semantics, delay and freshness. In summary, our analysis concludes the following:

(i) Impossibility of optimal order-preserving reads. Ensuring optimal reads (Section 10.5) is not possible under Order-Preserving or Atomic Visibility (Section 11.2).

(ii) Order-Preserving Visibility with minimal delay and concurrent freshness. Order-Preserving Visibility can ensure concurrent freshness at minimal delay (Section 11.3.2).

(iii) Atomic Visibility with minimal delay forces stable freshness. To ensure minimal delay, Atomic Visibility forces transactions to read from a *stable snapshot*, i.e., a snapshot consisting of updates known to have committed in the past (Section 11.3.3).

(iv) Consistent reads with latest freshness. To guarantee reading the freshest data, Order-Preserving and Atomic Visibilities require reads and updates mutually exclusive (Section 11.4).

11.1 Notation and Definitions

Notation. A committed update transaction creates a new version of the data items it updates. For some data item (or object) $x \in \mathcal{X}$, where $\mathcal{X}$ is the universe of object identifiers, we denote a version $x_v \in V$, where V denotes the universe of versions. We assume an initial state $\perp$ consisting of a initial version $x_\perp$ for every $x \in \mathcal{X}$. If versions follow a partial or total order $O = (V, \prec)$, we say a version x_i is more up-to-date (or fresher) than a version y_j when $y_j \prec x_i$.

The database is partitioned, i.e., its state is divided into $P \geq 1$ disjoint subsets, where all the versions of a given object belong to the same partition. Throughout the text, we use the terms partition, server and storage server interchangeably.

Definitions. We define the three types of snapshots introduced in Section 10.2 formally:

Definition 2 (Committed snapshot)**.** A committed snapshot S is any subset of V that includes exactly one version of every object $x \in \mathcal{X}$. $\mathcal{S}$ denotes the set of all committed snapshots.

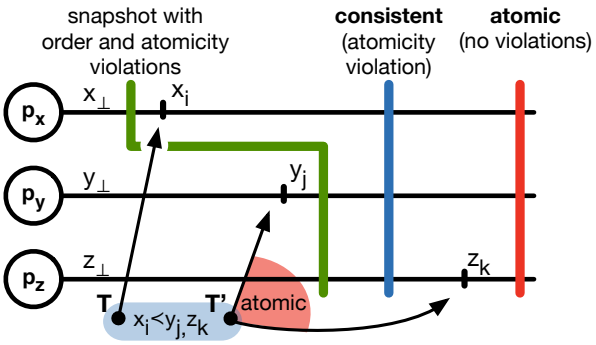


Figure 11.1: The three snapshot guarantees

Definition 3 (Order-Preserving Snapshot)**.** Given a partial or total order of versions $O = (V, \prec)$, a committed snapshot $S_O \in \mathbb{S}$ preserves O if $\forall x_i, y_j \in S$, $\nexists x_k \in V$ such that $x_i \prec x_k \prec y_j$. Intuitively, there is no gap in the order of versions visible in an order-preserving snapshot. We denote $\mathbb{S}_O \subseteq \mathbb{S}$ the set of committed snapshots preserving order O.

Definition 4 (Atomic Snapshot)**.** Given an order O, an order-preserving snapshot $S_A \in \mathbb{S}_O$ is atomic if $\forall x_i, y_j \in V$ such that x_i, y_j were written by the same transaction, if $x_i, y_k \in S_A$ then $y_k \not\prec y_j$, i.e., it disallows "broken reads". We denote $\mathbb{S}_\mathcal{A}$ the set of atomic snapshots for order O, $\mathbb{S}_\mathcal{A} \subseteq \mathbb{S}_O$.

Definition 5 (Snapshot guarantee)**.** Given some order O, we say that a read protocol guarantees Committed (, Order-Preserving or Atomic) Visibility if it guarantees that every transaction reads from a Committed (, Order-Preserving, or Atomic, respectively) snapshot.

We illustrate the three types of snapshots in Figure 11.1. The figure shows a system consisting of three partitions, p_x, p_y, and p_z, each storing a single object x, y, and z, in an initial state $x_\perp$, $y_\perp$, $z_\perp$, respectively. Two transactions T and T' have committed updates in order $x_\perp, y_\perp, z_\perp \prec x_i \prec y_j, z_k$. T updates only partition p_x, whereas T' updates p_y and p_z atomically. The figure highlights three possible snapshots. Under Atomic Visibility, only the atomic snapshot is admissible, precluding both order violation (both T and T''s updates are included) and read skew (as both y_j and z_k are included). Under Order-Preserving Visibility, the atomic and the order-preserving snapshots are both admissible. The latter precludes order violations, but not read skews (e.g., the snapshot includes y_j from transaction T', and not z_k). Finally, under Committed Visibility, all three depicted snapshots are admissible because order violations and read skews are allowed. The snapshot at the left of the picture exhibits two anomalies: an order violation, and a read skew. The order violation occurs by reading $x_\perp$ and y_j, as $x_\perp \prec x_i \prec y_j$, and x_i is not read. The read skew occurs by reading $z_\perp$ and y_j, as y_j was created atomically with z_k, which is not read.

11.2 Impossibility of optimal reads under ordered visibility

Proposition 5. A read protocol that guarantees Order-Preserving (or Atomic) Visibility cannot ensure optimal (delay and freshness) reads.

Proof. We prove this proposition by contradiction. Assume that there exists a read-optimal protocol that guarantees Order-Preserving (or Atomic) Visibility, w.r.t. order $O = (V, \prec)$. Consider the execution in Figure 11.2 where, initially, partition p_x stores $x_\perp$ and p_y stores $y_\perp$. Two transactions T_u and $T_{u'}$ write x_k at p_x and y_j at p_y respectively, establishing the following order: $x_\perp, y_\perp \prec x_k \prec y_j$. For instance, under causal order, this can result from an execution where a transaction reads $x_\perp$, and updates x, creating x_k, and later another transaction reads x_k and updates y, creating y_j. A T_r, running concurrently with T_u and $T_{u'}$, reads objects x and y in parallel from p_x and p_y. T_r reaches p_x before the creation of x_k, and p_y after the creation of y_j. To satisfy read optimality, partitions must reply immediately with the latest version they store, namely $x_\perp$ and y_j, observing an order violation[1] . Contradiction. ■

11.3 What freshness is compatible with minimal delay?

In this section we explore which are the maximum freshness degrees achievable for each snapshot guarantee, under the requirement of minimal delay.

11.3.1 Optimal reads under Committed Visibility

Proposition 6. A read protocol that guarantees Committed Visibility can be optimal.

Proof. Committed Visibility imposes no restrictions to the committed versions a transaction can read. Therefore, to serve a request under this model, a partition can reply immediately with the latest object version it stores. ■

11.3.2 Order-Preserving visibility and concurrent freshness

Proposition 7. A read protocol that guarantees Order-Preserving Visibility and minimal delay can ensure concurrent (Sector 2 of Figure A.1) or stable freshness, but not latest.

We prove this proposition by sketching a read protocol with such characteristics, followed by a correctness proof. In Chapter 13, we present a protocol with these characteristics.

Consider a protocol that orders its updates following some order $O = (V, \prec)$, and where reads preserve O. When a read transaction starts, the protocol assigns it an O-preserving *stable* snapshot S_O (Section 10.4). Read requests are sent to their corresponding partition in parallel. A partition can reply immediately with the version in S_O or with a more up-to-date version that is

[1] A similar situation occurs with the execution of Figure 11.3, where reading $x_\perp$ and y_j results in a read skew.

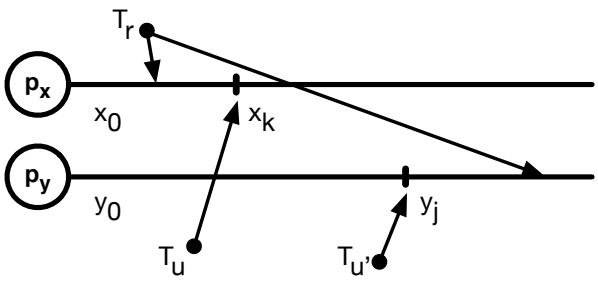


Figure 11.2: A read transaction executes concurrently with two update transactions at two partitions

compatible with S_O. An object version y_j is compatible with a given order-preserving snapshot S_O if replacing version $y_s \in S_O$ by y_j results in an order-preserving snapshot. Formally:

Definition 6 (Compatible version)**.** Given an order $O = (V, \prec)$, a version $y_j \in V$ and an order-preserving snapshot S_O, an object version $y_j \notin S_O$ is compatible with S_O, if $\forall x_i \in S_O$, $\nexists x_k \in V$ such that $x_i \prec x_k \prec y_j$.

Lemma 1. Given an order-preserving snapshot S_O, replacing any number of versions $x_o \in S_O$ by $x_i \notin S_O$, such that x_i is compatible with S_O, results in an order-preserving snapshot $S_{O'}$.

Proof. Proof. Assume by contradiction that the resulting snapshot $S_{O'}$ is not order-preserving w.r.t. order $O = (V, \prec)$. According to Definition 3, this implies that $\exists x_i, y_j \in S_{O'}, x_k \in V : x_i \prec x_k \prec y_j$. Since S_O is order-preserving, if the versions returned by read partitions were those in S_O, i.e., x_o and y_o, no inconsistency could have been created. Now consider the case where only one compatible version with S_O, e.g., y_j, is more up-to-date than $y_o \in S_O$ ($y_o \prec y_j$). By Definition 6, $\nexists x_k \in V : x_o \prec x_k \prec y_j$. Finally, assume that both $x_i, y_j \notin S_O$ are more up-to-date compatible versions of objects x and y. As they are compatible with S_O, by Definition 6, *(i)*$\nexists x_k : x_o \prec x_k \prec y_j$ and $\nexists y_l : y_o \prec y_l \prec x_i$. Moreover, we know that *(ii)*$x_o \prec x_i$ and $y_o \prec y_j$. *(i)* and *(ii)* imply $y_j \not\prec x_i$ and $x_i \not\prec y_j$: x_i and y_j are concurrent (incomparable in O). Therefore, there cannot exist $x_k : x_i \prec x_k \prec y_j$. Contradiction. ■

Lemma 2. The above protocol guarantees Order-Preserving Visibility.

Proof. This follows directly from Lemma 1. ■

Lemma 3. The above protocol allows concurrent freshness.

Proof. We prove this lemma by describing a sample execution. Assume transaction T_r starts with stable snapshot $S = \{x_o, y_o\}$. T_r sends a read request for objects x and y to partitions p_x and p_y respectively. Concurrently, update transactions create versions x_u and y_v establishing the following order between them: $x_o, y_o \prec x_u \prec y_v$. T_r's request arrives to p_x after x_u and y_v are committed. By Definition 6, p_x can reply with x_u, a more up-to-date version. However, p_y can

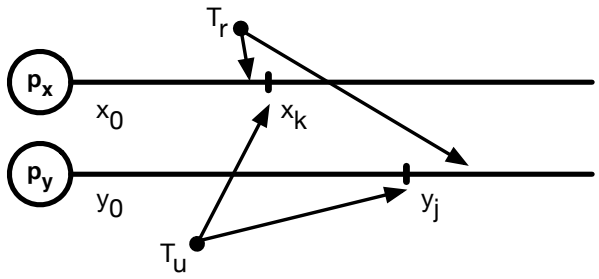


Figure 11.3: A read transaction executes concurrently with an atomic update transaction at two partitions

only reply with y_o, as y_v is not compatible with S ($\exists x_u \in V : x_o (\in S) \prec x_u \prec y_v$). As x_u is committed by an update transaction concurrent to T_r, this execution exhibits concurrent freshness. ■

Lemma 4. The above protocol guarantees minimal delays

Proof. The protocol reads versions in parallel. In the absence of fresher committed updates than those in S_O, a partition can reply immediately with versions belonging to S_O, which is stable and, therefore, already committed. In the presence of fresher and compatible committed updates, a partition can reply to a request with those, immediately. ■

Proof of Proposition 7. This follows directly from Lemmas 1, 4, 2 and 3.

11.3.3 Minimal-delay Atomic Visibility requires stable freshness

Proposition 8. A minimal-delay read protocol that guarantees Atomic Visibility requires stable freshness.

The intuition is that, due to the minimal-delay requirement, a partition receiving a read request from a transaction T_r cannot know whether other partitions accessed by T_r are returning updates of a concurrent update transaction T_u or not, which forces it to read from a stable snapshot to avoid Read Skews.

Proof. Assume by contradiction that there exists a minimal-delay protocol that guarantees Atomic Visibility and allows a transaction to read updates committed by other concurrent transactions (concurrent freshness). Consider the example execution in Figure 11.3. A transaction T_r sends parallel requests to read objects x and y from partitions p_x and p_y respectively. A concurrent transaction T_u commits versions x_k and y_j. Assume that T_r's request reaches p_y after T_u commits. By Definition 4, p_y can return y_j only if it is certain that T_r will read x_k from p_x. Due to the minimal-delay requirements, p_x does not have access to such information, since reads can be executed in parallel and no extra communication among partitions is allowed. Given that T_r can reach p_x before T_u commits x_k, p_y cannot risk returning y_j, and must ignore T_u. Therefore, a partition can only return a version of an update transaction T_u if it knows T_u had committed at

all its updated partitions by the time T_r sent its read requests. This implies that T_r has to read from stable snapshot, which contradicts our assumptions. ■

11.4 What is possible under latest freshness?

Proposition 9. Order-Preserving (and Atomic) Visibility require mutually-exclusive reads and updates to guarantee latest freshness.

Lemma 5. It is not possible to guarantee Order-Preserving or Atomic Visibility with latest freshness under bounded delay.

Proof. Consider again the sample execution of Figure 11.2 (where $x_\perp, y_\perp \prec x_k \prec y_j$). To ensure latest freshness, partitions must reply to read requests with the latest committed version they store. If p_x returned $x_\perp$ and p_y returned y_j, T_r would observe an inconsistent result (by missing x_k). The protocol could retry reading from p_x to read x_k, thus ensuring reading a version compatible with y_j. If such request arrived to p_x before T_u created x_k, p_x could block until x_k was applied to read it. During the blocking period, a concurrent update transaction may have written a new version x_m such that $y_j \prec y_w \prec x_m$. To satisfy latest freshness, p_x would be forced to reply with x_m, inconsistent with the version read from p_y: y_j. If updates are not stopped, this situation can repeat itself indefinitely, making reading with bounded delays impossible. ■

Lemma 6. A read protocol can ensure Order-Preserving (and Atomic) Visibility and latest freshness by enforcing mutually-exclusive reads and updates.

Proof. We prove this lemma by following the proof of Lemma 5. In the execution of Figure 11.2, T_r can retry indefinitely reading the latest versions of x and y until the results belong to an order-preserving snapshot. The equivalent holds for building an atomic snapshot under the execution of Figure 11.3. ■

Proof of Proposition 9. This follows directly from Proposition 5, and Lemmas 6 and 5.

11.5 Isolated reads with bounded delays and concurrent freshness.

Lemma 7. A read protocol can ensure Order-Preserving (and Atomic) Visibility and concurrent freshness under bounded delays.

Proof. Consider again the execution in Figure 11.2 where read transaction T_r executes concurrently with update transactions T_u and $T_{u'}$. If p_x returns $x_\perp$ and p_y returns y_j, it is possible to issue a second round to force p_x to return x_k, which would ensure Order-Preserving Visibility and concurrent freshness. The same holds for ensuring Atomic Visibility in the example execution in Figure 11.3. ■

As we will see in Chapter 16, many existing systems exhibit these characteristics.

Chapter 12

Unexplored Isolation Levels

An isolation level restricts the interleaving of the operations of one transaction with those of others, in order to forbid certain anomalous behaviours. Under multi-version concurrency control, a transaction reads from a snapshot. When committing, a commit protocol checks that the operations of the transaction have interleaved with the operations of other transactions only in ways permitted by the isolation level it implements (Chapter 3.3.2.2). In Chapter 10.2, we discussed the three snapshot guarantees considered for the trade-off. In this chapter, we discuss some new isolation levels that result from the combinations of those snapshot properties with different commit guarantees. Table 12.1 shows the properties of the combinations of read and commit guarantees. Order-Preserving Visibility can be combined in unexplored ways. **TCC$^-$** is the combination of Order-Preserving Visibility with non-serialised atomic updates. **PSI$^-$** combines Order-Preserving Visibility with write serialisation (or write-write conflict) checks. Changing Update Serialisability (US) so that read-only transactions ensure Committed and Order-Preserving visibility results in two models, which we call **CV-US** and **OP-US**, respectively. Table 12.2 compares the anomalies of several models.

12.0.1 CV-US and OP-US

Under Update Serialisability (US), only update transactions are serialisable. Read-only transactions read from Atomically-consistent snapshots that do not form a monotonic order. CV-US

Snapshot / Termination	**Partial Order of Updates**			**Total Order of Updates**	
	Not Ser.	**Ser. Writes**	**Ser. R. and W.**	**Ser. Writes**	**Ser. R. and W.**
Committed	RC	RC$^+$	***CV-US***	RC$^+$	
Order-Preserving	***TCC$^-$***	***PSI$^-$***	***OP-US***	***PSI$^-$***	S
Atomic	TCC/RA	PSI	US	SI	

Table 12.1: Combination of snapshot and termination guarantees - In bold: combinations not previously studied

relaxes the guarantees of read-only transactions to committed visibility and, OP-US, from relaxing reads to Order-Preserving visibility.

12.0.2 TCC$^-$

Under Transactional Causal Consistency (TCC), a transaction reads from an atomically-consistent snapshot. Both reads and updates can be concurrent (they are not serialised). TCC$^-$ weakens TCC's read guarantees to Order-Preserving Visibility. Therefore, when compared to TCC, TCC$^-$ further allows the read-skew anomaly. On the positive side, as the results of the trade-off show, this relaxation positions TCC$^-$ in a better position with respect to latency and freshness. In the next chapter, we introduce a protocol offering these guarantees. In Chapter 14, we evaluate this protocol in terms of latency, throughput, and freshness.

12.0.3 PSI$^-$

Under Parallel Snapshot Isolation (PSI), a transaction reads from an atomically-consistent snapshot, and updates undergo serialisation checks. PSI$^-$ results from weakening the read guarantees of PSI to Order-Preserving Visibility. Equivalently to TCC and TCC$^-$, when compared to PSI, PSI$^-$ further allows the read-skew anomaly. By the trade-off results, this relaxation positions PSI$^-$ better with respect to latency and freshness.

Fresh reads has the potential to reduce the abort rate of update transactions. Implementations of PSI (and SI) perform serialisation checks on a transaction's updates by verifying that other transactions have not updated the same objects concurrently. For each updated object, they check there exists no updates that have not been observed by the transaction's snapshot. If a concurrent update is detected, the transaction aborts. Reading up-to-date values of the updated objects is, therefore, required to commit a transaction.

An interesting observation is that under a similar implementation, a protocol ensuring PSI$^-$ would disallow Read Skews over among the objects a transaction updates. This occurs because of serialisation checks disallowing the existence of newer versions of updated objects. An experienced programmer desiring the latency/freshness properties of Order-Preserving reads could issue updates on the particular objects it requires reading atomically. Unfortunately, we have not experimented with algorithms implementing PSI$^-$.

	CP Models									AP Models				
Anomaly / Model	**SS**	**S**	**US**	***OP* US**	***CV* US**	**SI**	**PSI**	***PSI*$^{-}$**	**RC+**	**TCC**	***TCC*$^{-}$**	**RA**	**RC**	**NI**
Dirty Read	x	x	x	***x***	***x***	x	x	x	x	x	x	x	x	-
Read Skew	x	x	x	**-**	**-**	x	x	**-**	-	x	-	x	-	-
Lost Update	x	x	x	***x***	***x***	x	x	x	x	-	-	-	-	-
Write Skew	x	x	x	***x***	***x***	-	-	x	-	-	-	-	-	-
Long Fork	x	x	-	-	-	x	-	-	-	-	-	-	-	-
Order violation	x	x	x	***x***	**-**	x	x	x	-	x	x	-	-	-
Time Violation	x	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 12.2: Anomaly comparison of new isolation levels

Chapter 13

Minimal-delay protocol design

In this section, we apply the trade-off analysis to protocol design. Motivated by the tight latency requirements of cloud services (Section 2.2), we modify Cure to derive three minimal-delay read protocols, called CV, OP and AV, which ensure respectively Read Committed, TCC$^-$, and TCC isolation.

Cure's reads ensure Atomic Visibility preserving causal order, bounded delay and concurrent freshness. Using the insights taken from the analysis, we remove the extra delays of Cure to achieve minimal-delay: one can either degrade read semantics or freshness. Each protocol occupies a different point in the three-way trade-off. Cure belongs in Sector 5 of Figure A.1. CV's reads ensure Committed Visibility and can achieve both minimal delay and latest freshness (Sector 3 in the figure). To provide Order-Preserving Visibility (required by TCC$^-$) with minimal-delay reads, the best possible freshness for OP is concurrent freshness (Sector 2). Similarly, to provide Atomic Visibility (required by TCC), AV requires stable freshness (Sector 1).

13.1 Cure recapitulation

Our base transactional protocol (Cure) ensures Transactional Causal Consistency (TCC) (causal-consistency and Atomic Visibility), bounded delays and concurrent freshness. It can be deployed in a multi-site setting, where transactions execute entirely within a site —each site stores a full replica of the state and all sites are equally partitioned— and replicate updates across sites asynchronously. We refer to partitions storing the same set of objects at different sites as *sibling* partitions.

The interface exposes two primitives to access data: *multi* reads and updates. Through these primitives, a client can group any number of read or update operations (a single multi-operation cannot include both), and execute them against the storage servers in parallel. A transaction can comprise any number of such multi-operations, followed by a commit instruction.

In order to read consistently, the protocol associates a snapshot to the transaction when it starts. Partitions reply to read requests with the version in the snapshot. To enable the above

mechanism, the store is multi-versioned. The protocol exhibits concurrent freshness and bounded delays as a partition might need to block waiting for concurrent update transactions to commit updates belonging to the transaction's associated snapshot, and in cases of clock skew between servers (explained in Section 6.2). Causal order information is encoded in a vector sized with the number of sites to provide a good balance between update visibility latency and throughput (Section 5.3.3). The protocol can be characterised as *Deferred Update Replication* (DUR), i.e., updates are buffered during a transaction's execution and only sent to storage servers at commit time [77], and applied atomically by means of a classic two-phase commit protocol.

13.2 Changes to the base protocol

In the base protocol, the snapshot associated to transactions is not necessarily stable. Thus, when a read request reaches a partition, the version satisfying the snapshot may not be yet available, forcing the partition to delay the reply until it is.

To minimise delays of update transactions, all protocols provide low-latency atomic updates across partitions by means of a two-phase commit (2PC) protocol using an optimisation where a transaction is considered committed after a successful prepare phase, when all involved partitions have confirmed they will commit. A client can then receive a response after a single round-trip of communication with updated partitions. This complicates recovery in cases of failures, which we do not address in this work.

In order to provide minimal-delay reads for CV, OP and AV; we have applied the following modifications:

- AV achieves minimal delays by degrading the base protocol's freshness: it ensures the snapshot assigned to transactions on init is stable.
- OP makes one further modification to achieve better freshness, it degrades visibility to Order-Preserving. A partition is allowed to return a more up-to-date compatible version (Definition 6, Section 11.3.2) with the stable snapshot when available.
- CV degrades read guarantees to Committed Visibility to attain optimal freshness by ensuring partitions always return the latest committed version.

13.3 Transaction execution overview

Setting and notation. The design of the protocols considers M fully-replicated sites. Every site partitions data into N partitions. All sites follow the same partitioning scheme. All algorithms share a general skeleton. In this section, we explain the algorithms points in common and, in §13.4, each algorithm's particularities. They involve two kinds of processes for executing a transaction: a *transaction coordinator (TC)*, and the *partitions*.

13.3.1 Transaction Coordinator (Algorithm 13.1)

This ephemeral process is started at the server that receives a client request to run a transaction. It lives throughout the execution of the transaction and terminates once the transaction commits (or aborts in case of a failure during an update transaction).

Init. A TC initialises (Lines 33-38) on a client's first read or update request. It initialises the write set for the transaction WS_T, the associated snapshot ss_T, the transaction's commit time ct_T, and the dependency vector clock dep_T (used for creating a causal order of updates).

Reads. When receiving a read request for a list of keys, i.e., object identifiers (Line 1), the TC groups keys by partition by calling the $GET\_PARTITIONS$ function (Line 4) and sends a read request to each partition in parallel (Line 6). Once it receives all responses (Line 7), it returns the read values to the client.

Updates and commit. When a client submits updates, the transaction coordinator buffers them (Line 17) in its write set. When the client calls commit (Line 20), if the transaction updates multiple partitions, the transaction coordinator coordinates a two-phase commit protocol among the updated partitions (Lines 23-30). In the prepare phase, the TC sends, in parallel, prepare messages with the updates in the write set to each updated partition (Line 24). Once it collects prepare responses from all participants, the TC replies to the client confirming that the transaction has committed, and sends the commit instruction to all partitions. The commit time is set to the maximum of the prepare times proposed by the participants.

The case where a transaction commits updates at a single partition avoids a second round of messages by collapsing the prepare and commit instruction in a single message. Such optimisation is not depicted in the algorithm. Aborts are only possible due to failures as a participant never proposes to abort a transaction. To simplify the explanation of the algorithm, we do not illustrate the abort path.

13.3.2 Partitions (Algorithm 13.2)

Partitions store versions of objects and reply to requests from transaction coordinators that access those objects.

Reads. When a partition receives a read request with a list of keys, it replies with the most up to date version of each requested object that satisfies the algorithms' visibility criteria (Lines 20-23).

Updates and commit. Partitions participate in two-phase commit instances coordinated by TCs. In the first phase, a partition receives a prepare message (Line 8), buffers the updates received in such message, and replies with a *prepared* message; a positive vote to commit the transaction. At this stage, these updates cannot be read by other transactions, as the transaction has not yet committed. Later, after a TC has collected the prepared messages from all participants, partitions receive a commit message (Line 15) confirming that the transaction commits. Each partition applies its updates, making them available to future readers. Identically to Cure,

prepare instructions are logged synchronously and commit instructions, asynchronously, for recovery (explained in Section 6.2).

Update propagation. Under replication, partitions propagate updates asynchronously to remote sites. Once an update transaction commits, an updated partition submits its updates to be sent to remote sibling partitions. This is done by a periodic batch process depicted in Algorithm 13.3, explained in the next section. When a partition receives a remote transaction's updates, it applies them locally and makes them available to future read operations.

13.4 Protocol particularities and correctness

In this section, we introduce the details of each introduced algorithm and discuss correctness.

Consistent Reads. OP and AV provide isolated reads. When a TC executes the first read (or update) of a transaction, it initialises the transaction's snapshot time to ss_T (the transaction's stable snapshot), which is used as a pivot to compute versions that satisfy the guarantees of each algorithm. ss_T is assigned the partition's stable vector SV_n^m, which denotes the latest stable snapshot known by the partition where the TC runs (Line 37). In Section 6.3, we describe how each partition maintains this vector.

A TCs includes ss_T in read requests sent to partitions (Line 6). When a partition receives a request, it responds with the newest version that complies with the requested snapshot, according to a $COND$ function defined per protocol:

- Under CV, $COND$ simply returns the latest committed version.
- Under AV, $COND$ ensures versions do not violate an atomic snapshot that preserves causal order. For each requested object version, AV returns the version belonging to the stable snapshot characterised by ss_T; i.e, the newest version with commit vector smaller than ss_T (Alg. 13.2, Line 23). Since the snapshot is stable, all transactions with $cv \leq ss_T$ have committed, and partitions can reply immediately. Moreover, this guarantees Atomic Visibility (the absence of fractured reads) as, under this protocol, all updates of a transaction commit with the same commit timestamp (explained later in this section).
- Under OP, $COND$ ensures versions belong to a causal-order-preserving snapshot. For this, partitions either return the version belonging to the stable snapshot characterised by ss_T or a more up-to-date version compatible with ss_T, if available. According to Definition 6, a version is compatible with a snapshot if it is not ordered after versions fresher than those in the snapshot. In this protocol, this means a version is compatible if its associated dependency vector is not larger than ss_T (Alg. 13.2, Line 22).

Causal order of updates. OP and AV require updates to be causally ordered. Transactions create object versions with a dependency vector dep, which indicates a version is ordered causally

after all versions with commit vector $cv \leq dep$. A commit vector cv is created by replacing, in dep, the entry of the site where the version was committed by its commit time ct. To establish a correct causal order, these algorithms must ensure a version's cv is larger than its dep. The transaction coordinator ensures this by picking a ct which is larger than the transaction dependencies ((in Alg. 13.1, Line 21). To ensure that dep is larger than the cv of its causal dependencies:

- Under OP, after receiving read responses, the TC updates the transaction's dependency vector dep_T to be the maximum commit vector of a read version (Alg. 13.1, Line 12). This is necessary as transactions can read versions committed with commit timestamp larger than the transaction's ss_T.
- AV's read algorithm ensures that a transaction will never read versions committed with a timestamp larger than the transaction's ss_T. Therefore, for AV, it suffices to assign ss_T to the transaction's dependency vector dep_T (Alg. 13.1, Line 37).

These protocols further ensure updates of a transaction commit with the same cv by choosing a transaction's ct as the maximum proposed time by an updated partition (Alg. 13.1, Line 27). This is used by AV's read protocol to ensure atomic visibility.

13.5 Stabilisation protocol

Under OP and AV, transactions use the knowledge of a stable snapshot to read consistently and with minimal delay. A background stabilisation protocol runs among all partitions in the system to compute stable snapshots at each site. Algorithm 13.3 describes this protocol. It includes the following steps:

1. When a partition commits a transaction, it adds the transaction's updates to to_send, the list of updates to be sent to sibling partitions at remote sites (Alg. 13.2, Line 18).
2. Periodically, updates in to_send are propagated in commit-time order (Line 9). When there are no updates to be sent, the partition sends a heartbeat message, used by siblings to advance their stable snapshots, explained as follows (Line 10).
3. Each partition maintains a vector vec_p with an entry per site. Each entry j indicates that a partition p of site i has applied locally all updates with commit time $ct \leq vec_p[j]$ from its sibling partition at site j. To ensure updates are sent to sibling partitions in commit ct order, the local entry of the vector is set to be smaller than the minimum prepared time of transactions committing locally (Line 3).
4. vec_p is exchanged among partitions belonging to the same site (Line 6).
5. SV_m^n, representing the latest stable snapshot, is computed as the minimum of all vec_p received (Line 13).

The protocol ensures the snapshot is stable as (i) no local partition will commit a transaction with commit time smaller than $vec_p[i]$, and (ii) no remote update will be further received from a remote site j with commit time smaller than $vec_p[j]$.

13.6 Causal consistency: session guarantees

13.6.1 Read your writes

The algorithms presented do not ensure the *read your writes* session guarantee, required by causal consistency. Under AV and OP, a client that executed an update transaction and in a subsequent transaction reads the objects previously updated may not observe them. The problem arises from the time it takes for an update to become stable (and included in a partition's stable snapshot).

Read your writes can be enforced by clients caching their latest updates. When receiving a read response, the client compares the version received with the cached one (if any). If the version cached is fresher, the one returned by the system is discarded. After a transaction finishes, a TC returns the latest stable vector it is currently aware of for clients to empty their caches accordingly.

13.6.2 Monotonic Reads

Under AV and OP, monotonic reads are ensured when clients connect always to the same server. Under failures, a client could connect to a server which's SV is behind that of the ss of its last transaction. This could lead to observing data less up to date than what was previously observed, thus violating the monotonic-reads session guarantee.

Monotonic Reads can be ensured by clients informing transaction coordinators of their latest ss when issuing transactions. For this, when a transaction commits, its TC should return the transaction's ss, and the client should issue new transactions by sending this information to new TCs. Upon receiving a request including a ss, a TC running at site n that detects that its SV is behind, it must ensure it catches up before proceeding. This can be done as follows: if TC runs at the same site as the latest transaction, it can update its SV to ss immediately, as the snapshot was previously observed as stable by another partition on the same site. If the TC runs at a different site, TC has to block until $SV \geq ss$.

Algorithm 13.1 Transaction Coordinator tc at site n

```
1:  function READ_OBJECTS(keys)
2:      If (!initiated) INIT( )
3:      result = ∅
4:      read_partitions_T=GET_PARTITIONS(keys)
5:      for all ⟨p, keys_p ⟩ ∈ partitions_T do
6:          send ⟨read, keys_p, ss_T⟩ to p
7:      for all ⟨p, keys_p ⟩ ∈ partitions_T do
8:          receive ⟨partition_result⟩ from p
9:          result = result ∪ {partition_result}
10:     If (protocol == OP)
11:         commit_vc=MAX_v(v.commit_vc ∈ result)
12:         dep_T=MAX_v(dep_T, commit_vc)
13:     return result.values
14:
15: function UPDATE_OBJECTS([⟨key, update⟩])
16:     If (!initiated) INIT( )
17:     WS_T = WS_T ∪ {[⟨key, update⟩]}
18:     return ok
19:
20: function COMMIT( )
21:     ct_T=dep_T[n] +1
22:     updated_partitions_T = GET_PARTITIONS(WS_T.keys)
23:     for all ⟨p, updates_p ⟩ ∈ updated_partitions_T do
24:         send ⟨prepare, updates_p, dep_T, ct_T⟩ to p
25:     for all ⟨p, keys_p ⟩ ∈ updated_partitions_T do
26:         receive ⟨prepared, time_p⟩ from p
27:         If (protocol ≠ CV) ct_T=MAX(ct_T, time_p)
28:     return ok
29:     for all ⟨p, updates_p ⟩ ∈ updated_partitions_T do
30:         send ⟨commit, ct_T⟩ to p
31:     TERMINATE( )
32:
33: function INIT( )
34:     WS_T=∅
35:     If (protocol == OP or AV)
36:         ct_T = ⊥
37:         dep_T = ss_T = GET_STABLE_VECTOR( )
38:         initiated = true
```

Algorithm 13.2 Partition m at site n p_m^n

```
1: upon receive ⟨read, keys,ss_T⟩ from tc
2:     result=∅
3:     for all k ∈ keys do
4:         v= newest k_i ∈ ver[k] : COND(k_i, ss_T)
5:         result = result ∪ {v}
6:     send ⟨result⟩ to tc
7:
8: upon receive ⟨prepare, upd, dep_T, ct_T⟩ from tc
9:     If (protocol ≠ CV)
10:        time = MAX(READ_CLOCK( ), ct_T)
11:        prepared = prepared ∪ ⟨tc, upd, dep_T, time⟩
12:    Else prepared = prepared ∪ ⟨tc, upd⟩
13:    send ⟨prepared, time⟩ to tc
14:
15: upon receive ⟨commit, ct_T⟩ from tc
16:    prepared = prepared \⟨tc, upd, dep_T, time⟩
17:    UPDATE_VERSIONS(⟨upd, dep_T, ct_T, n⟩)
18:    to_send=to_send ∪ {⟨upd, dep_T, ct⟩}
19:
20: function COND(k_i, dep_T)
21:    If (protocol==CV) return true
22:    If (protocol==OP) return k_i.dep ≤ dep_T
23:    Else return k_i.cv ≤ dep_T
24:
25: function UPDATE_VERSIONS(upd, dep_T, ct, n)
26:    for all ⟨k,val⟩ in upd do
27:        If (protocol = CV) ver[k] = {val}
28:        Else ver[k] = ver[k]∪ {⟨val, dep_T, ct, n⟩}
```

Algorithm 13.3 Stabilisation for AV and OP at p_m^n

```
1:  periodically
2:      If (prepared ≠ ∅)
3:          stable_n = MIN(time ∈ prepared) - 1
4:      Else stable_n = READ_CLOCK( )
5:      vec_p[n] = stable_n
6:      send ⟨stable, vec_p⟩ to p_k^n, ∀k ∈ P, k ≠ m
7:      If (to_send ≠ ∅)
8:          for all ⟨upd_p, dep, ct⟩ ∈ send do
9:              send ⟨updates, upd_p, dep, ct⟩ to p_m^j, j ≠ n
10:     Else send ⟨heartbeat, stable_n⟩ to p_m^j, j ≠ n
11:
12: upon receive ⟨stable, vec_p⟩ from all p_k^n, k ≠ m
13:     SV_m^n=MIN_v(vec_p), # ∀p_m^j
14:
15: upon receive ⟨updates,updates, dep, ct⟩ from p_m^j
16:     UPDATE_VERSIONS(updates, dep, ct, n)
17:     vec_p[j] = ct
18:     # update known committed transactions from site j
19:
20: upon receive ⟨heartbeat,stable_j⟩ from p_m^j
21:     vec_p[j] = stable_j
22:     # update stable time from site j
```

Chapter 14

Evaluation

We empirically explore how the results of the three-way trade-off in Chapter 11 affect real workloads. We evaluate Cure and the three minimal-delay protocols presented in Chapter 13: CV, OP and AV.

14.1 Implementation

The implementation of the protocols was done in the same environment as Cure (Section 7.1), on the Antidote database.

Under CV, objects are single-versioned. Objects in OP, AV and Cure are multi-versioned. Each key stores a linked list of recent updates. Old versions are garbage collected using the same mechanism as Cure uses (Section 3.5.2.3).

14.2 Setup

Hardware. All experiments were run on a cluster located in Rennes, France, on the Grid5000 [51] experimental platform using fully-dedicated servers, where each server consists of 2 CPUs Intel Xeon E5-2630 v3, with 8 cores/CPU, 128 GB RAM, and two 558 GB hard drives. Nodes are connected through shared 10 Gbps switches. The ping latency measured within the cluster during the experiment was approximately 0.15 ms.

Configuration. Within the cluster, we configured two logical sites consisting of 16 machines each. Each site is comprised of 512 logical partitions, scattered evenly across the physical machines. Nodes within the same DC communicate using the distributed message passing framework of Erlang/OTP running over TCP. Connections across separate DCs use ZeroMQ sockets [6], also over TCP, where each node connects to all other nodes to avoid any centralisation bottleneck. The stabilisation protocol is run every 10 ms under OP, AV, and Cure, the same configuration used for the evaluation of Cure (Section 7).

Workload generation. The data set used in the experiments includes 100k keys per server, totalling 1.6 million keys per site. Objects are registers with the last-writer wins (LWW) policy [54], where updates generate random 100-byte binary values. All objects were replicated at all sites. A custom version of Basho Bench is used to generate workloads [1]. Google's Protocol Buffer interface is used to serialise messages between Basho Bench clients and Antidote servers [34]. To avoid across-machine latencies, two instances of Basho Bench run at each server, which issue requests to the Antidote instance running on it. Each instance of the benchmark was run for two minutes, the first minute being used as a warm-up period. A variable number of clients repeatedly run read-only and update transactions. Unless stated otherwise, read and update operations within a transaction select keys using a power-law distribution, where 80% of operations are directed to 20% of keys.

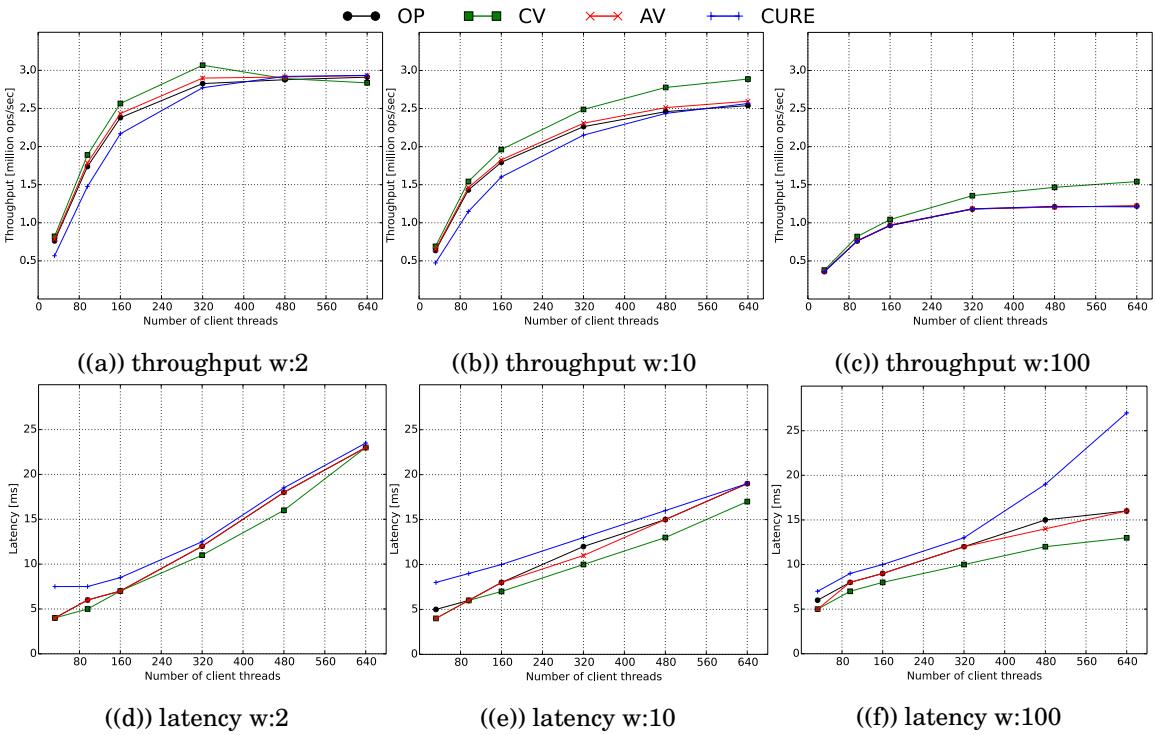
14.3 Experiments

We run all experiments in a single logical data centre, and in two. We observed very similar results under both configurations. In what follows, we only present the results of the experiments with two data centers.

We expect to observe a similar latency response for all minimal-delay protocols, and a slight degradation for Cure, which may block for a small amount of time under clock skew or due to not-yet-committed concurrent transactions. We expect to observe higher throughput for CV, as it does not incur the overheads of multi-versioning: traversing version lists to find a version compatible with a given snapshot, and garbage collecting versions. Moreover, we expect to observe OP, which offers Order-Preserving visibility to exhibit significantly better freshness than Cure and AV, which implement the strongest Atomic Visibility.

Workloads. We run experiments under two workloads which run different read-only transactions. Under the first workload, a read-only transaction reads a number of objects in parallel, in a single round, i.e., in a single call. In the second workload, we try to mimic Facebook's multi-read operations, by issuing read-only transactions that make many calls, each reading a number of objects in parallel.

Under both workloads, each client repeatedly executes a read-only transaction followed by an update transaction in a closed-loop (zero thinking time). we vary the number of client threads and measure how latency, throughput, and staleness change as load is added to the system. Throughput is measured in operations per second, where each operation is a read or write in a transaction. We measure staleness as follows. Upon treating a read request for a given object, a partition asynchronously logs the number of versions it needs to skip to guarantee a given isolation property. For Cure, a partition also logs the cases where it has to wait due to clock skew or for other transactions to commit. We process these logs offline.



((a)) throughput w:2 ((b)) throughput w:10 ((c)) throughput w:100

((d)) latency w:2 ((e)) latency w:10 ((f)) latency w:100

Figure 14.1: Single-shot read-only-transactions (100 read ops/txn)

Single-shot read-only transactions. A single-round transaction performs 100 reads in parallel. Update transactions perform 2, 10 or 100 updates, generating an update rate of approximately 2, 10 and 50% respectively.

Facebook-like read-only transactions. The multi-round experiment mimics the Facebook social network, where transactions read thousands of objects in tens to dozens of rounds, and updates represent 0.2% of the workload [33]. In our synthetic Facebook-like workload, a read-only transaction executes 10 rounds of 100 parallel reads each (totalling 1000 reads). Update transactions perform 2, 100 or 1000 updates, generating an update rate of approximately 0.2, 10 and 50% respectively.

14.3.1 Single-shot read-only transactions

We performed multiple runs of the experiment with a increasing number of client threads. A point on a curve represents the result of a particular run.

Throughput. We measure throughput as the total number of operations of both read-only and update transactions. Each graph shows the results under a different update rate. We observe that:

- Under all workloads, throughput increases when increasing the number of client threads, up to a point where the system works at full capacity, and adding more client threads does not result in more throughput. CV outperforms all other protocols due to the lightweight implementation enabled by its weak semantics: objects are single versioned and there is no overhead for ensuring read isolation.

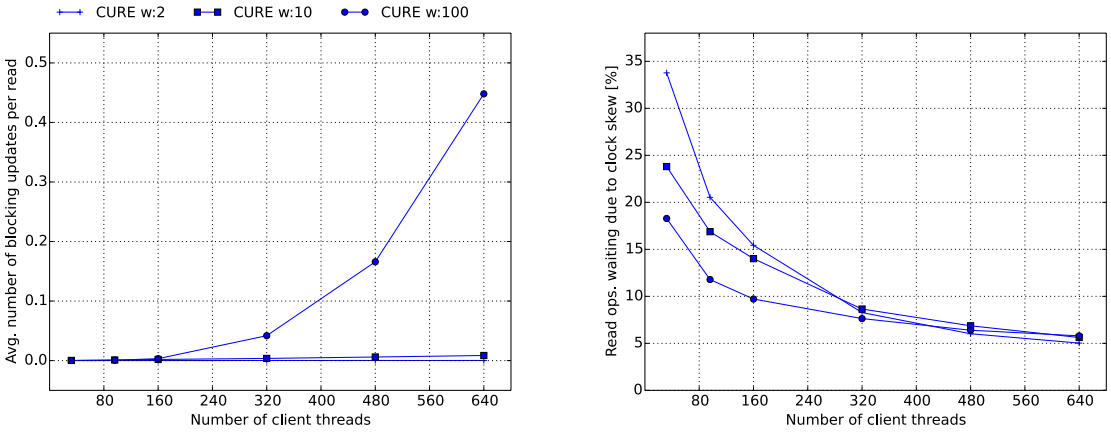
- Under a low update rate (**Figure 14.1(a)**), the gap between CV and the remaining protocols is small, and we observe that all protocols behave similarly. This occurs as, at low update rate, (i) version lists grow slowly, (ii) protocols are able to read the most up-to-date-version of each object most of the times (as we will see in the freshness discussion) and (iii) garbage collection of versions is less expensive. In this graph, under small number of client threads, we observe that Cure presents the smallest throughput. This happens because Cure requires blocking under scenarios of clock skew between servers. This slightly hampers the latency of its read-only transactions. As under small numbers of client threads the system is not working at its maximum capacity, this results in a degradation of throughput generated by blocked threads. As we increase the number of client threads, this effect dissipates, as larger number of clients saturate the system's capacity.

- Under approximately 10% of writes (**Figure 14.1(b)**), we observe that the gap between CV and the other protocols grows to a difference of up to 1.12X under high number of client threads. This happens due to the effects of handling and garbage-collecting multiple object versions to read under the protocols that enforce isolation. Under this workload and small number of client threads, we still observe that Cure presents the smallest throughput response due to the effect of clock skew between servers.

- Under 50% of writes (**Figure 14.1(c)**), all protocols degrade their overall throughput. We observe that the throughput gap between CV and the rest of the protocols is large — of up to approximately 1.35X for high number of client threads— because of the heavy-weight version management the other protocols incur. The effects of Cure's blocking scenarios is not visible, as a larger portion of operations are updates and the system behaves closer to its capacity limits.

Latency. Figures 14.1(d), 14.1(e) and 14.1(f) show the latency response of read-only transactions under 2, 10, and 100 writes per update transaction respectively. All systems exhibit a similar trend: increasing the number of client threads causes an increase in latency, as the system must handle higher load. CV exhibits the lowest latency, by a small difference with respect to OP and AV, which are latency optimal and behave similarly. This happens as CV does not incur overheads for searching for a compatible version.

Cure's Latency. The design of Cure is not latency optimal. Figures show the following:



((a)) Wait due to committing update

((b)) Wait due to clock skew

Figure 14.2: Cure blocking scenarios - Single-shot read-only-transactions (100 read ops/txn)

- For low update rates, Figures 14.1(d) and 14.1(e) show that, under small number of client threads, Cure exhibits extra (up to 1.9X) latency due to clock skew between servers. We plot, in **Figure 14.2(b)**, the percentage of read operations that blocked due to clock skew under Cure. This effect is frequent under small number of client threads, and dissipates as the system becomes more loaded. Under high load, the time it takes to process a received read-request message is larger, and during this time, lagging clocks can catch up.

- At high update rate (**Figure14.1(f)**), read operations in Cure wait for update transactions to commit frequently. This causes the latency gap between this protocol and the remaining ones when the number of client threads is large. **Figure 14.2(a)** shows how, when keys become highly contended (i.e., at high update rate and number of client threads), waiting for an update operation becomes frequent. At maximum contention —640 threads and 50% of updates— a read operation waits, on average, for 0.45 update operations to finish or, equivalently, each transaction waits for an average of 45 updates to commit.

Freshness. Figures **14.3(a)**, **14.3(b)**and **14.3(c)** show the freshness response as the number of client threads increases, for different update rates. Plots display the percentage of read operations that returned the most up-to-date version available at the contacted server. CV is not present in the figures as it always returns the latest version. Figures **14.3(d)**, **14.3(e)**and **14.3(f)** display a CDF showing how stale a read version is under 480 client threads for each update rate. We observe that:

- OP outperforms the other protocols under all workloads. Its freshness response remains nearly constant: over 99.8% of reads observe the latest version under all configurations. This shows that its design, which allows for concurrent freshness, allows this protocol to behave nearly optimally.

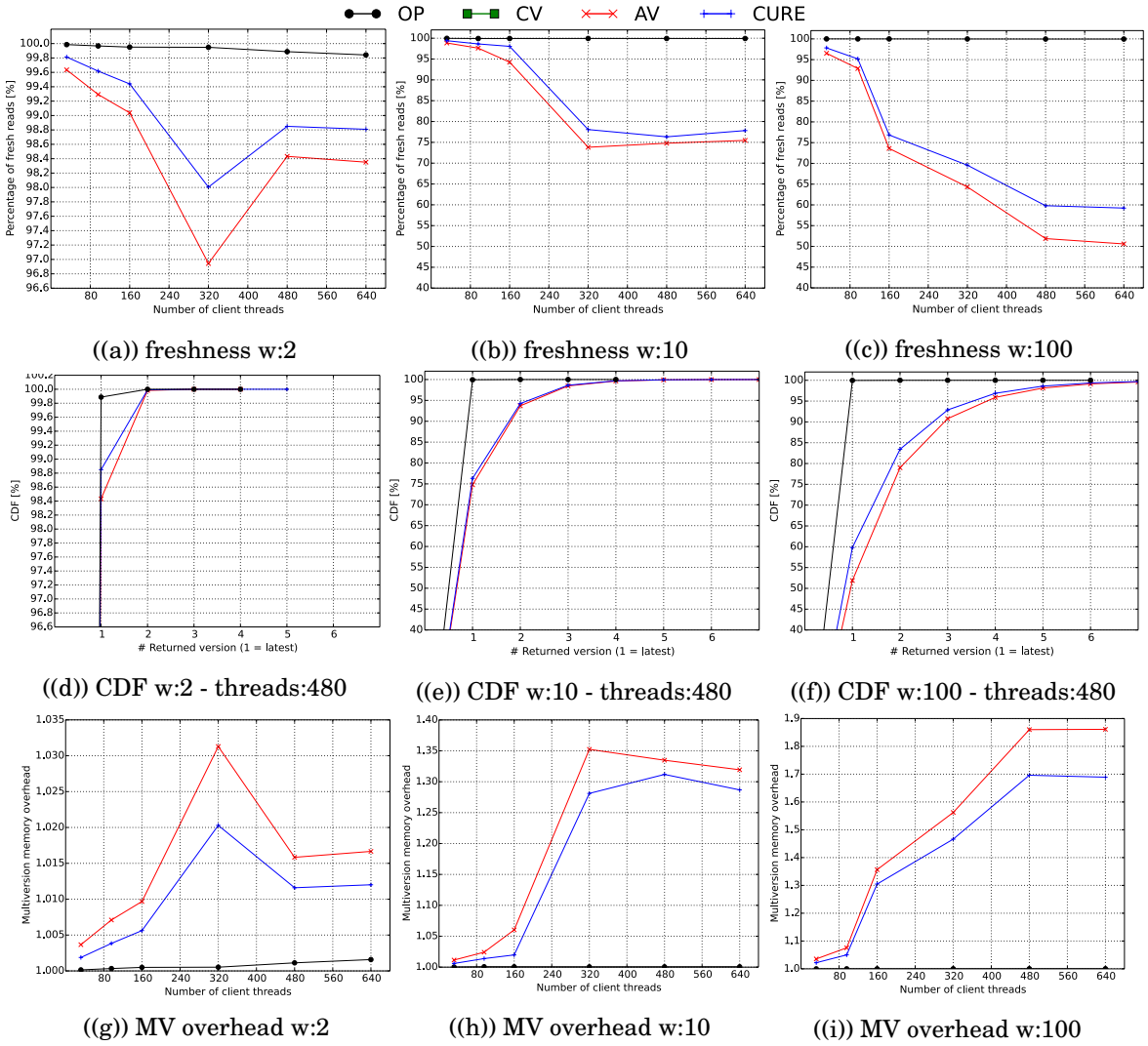


Figure 14.3: Freshness of single-shot read-only-transactions

- Under the 2%-writes workload (Figure **14.3(a)**), OP exhibits, 0.2% of stale reads in the worst case, while Cure 2% (10X), and AV 3% (15X). Figure **14.3(d)** shows how fresh reads were under 480 client threads. All protocols read, most of the times, the latest or second most recent version. OP read, in the worst case, the third most recent version, while the remaining two protocols, the fourth. Cure exhibits, under the same conditions, 1.2% (6X) and, AV, 1.8% (9X) stale reads, meaning that potentially every transaction observes stale versions.

- Under 10% of writes (Figure **14.3(b)**), OP does not further degrade its freshness, and reads observe the same percentage of stale versions, whereas freshness degrades significantly for AV, which shows 25% (125X) stale reads in the worst case, and Cure, which shows 22% (110X). Figure **14.3(e)** shows that OP read mostly fresh versions and, in the worst case, the

fourth most-recent version. Cure and AV show a higher frequency or returning the second (≈ 20%), third (≈ 5%) and fourth (≈ 2%) most recent versions. As each read transaction executes 100 reads, this means that this potentially affects every transaction. In the worst case, these two protocols read the 12th-to-latest version, not shown in the picture to display them more clearly.

- Under 50% of writes (Figure **14.3(c)**), we observe the biggest difference between all protocols: OP still suffers from up to 0.2% of stale reads, while Cure from up to 41% (205X) and AV 49% (245X). Figure **14.3(f)** shows that, for 480 threads OP read, in the worst case, the 6th-to-latest version. Cure the 18th and AV the 19th. Cure and AV frequently show significantly stale versions, up to the sixth (≈ 2%) version is potentially observed by every transaction. The oldest version returned by AV was the 19th to latest, and by Cure, the 18th to latest.

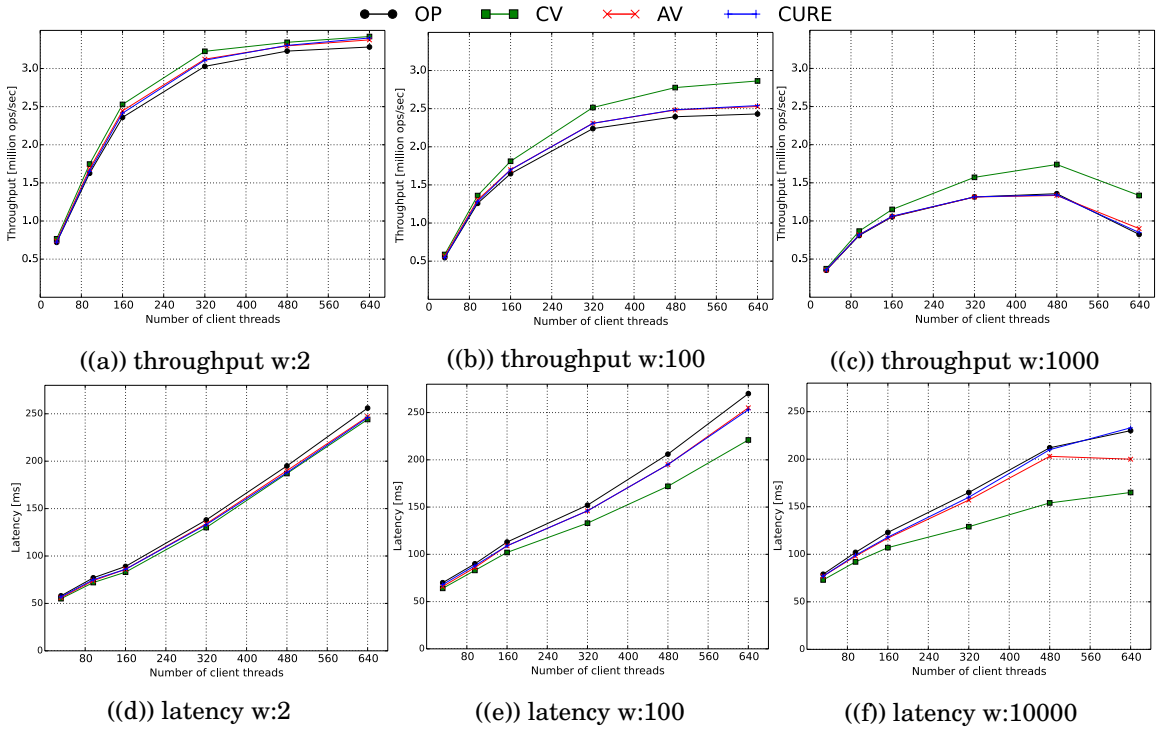
Multi-version overhead. We compute the multi-version overhead as the extra work required, for each read operation, to find and store a version that respects a transaction's required isolation level, with respect to a protocol like CV, which incurs no overhead. For instance, under this metric, a read that returns the second-to-latest version has an overhead of 2X over returning the latest. Reading the third to latest has an overhead of 3X, etc. This metric is computed in practice as the area over the lines of the CDFs. We compute, for each workload this metric as the overall overhead observed during the entire execution. Figures 14.3(g), 14.3(h) and 14.3(i) show the results under this workload.

The graphs exhibit a very similar trend to the freshness results, with the difference that freshness result do not make any distinction on the degree at which values are stale. For all workloads, OP shows a very low overhead, under 1.002X over an optimal protocol. CV presents a maximum overhead of 1.03X, 1.35X, and 1.87X under 2, 10 and 100 updates per transaction, respectively, while Cure 1.02X, 1.31X and 1.7X.

Conclusion. Under this workload, we have observed the effects of the three-way trade-off in action. Strengthening the semantics from Committed to Order-Preserving visibility incurs a negligible overhead in terms of latency and freshness. However, strengthening the semantics to Atomic Visibility penalises freshness significantly. Both protocols we have experimented with exhibited a high degradation in freshness. Cure exhibits better freshness than AV at a latency cost, which increases with contention.

14.3.2 Facebook-like read-only transactions.

We perform the same analysis under the Facebook-like workload. Figures 14.4, 14.5 and 14.6 show the results. Generally, results follow the same trend as those of single-shot read-only transactions.



((a)) throughput w:2 ((b)) throughput w:100 ((c)) throughput w:1000

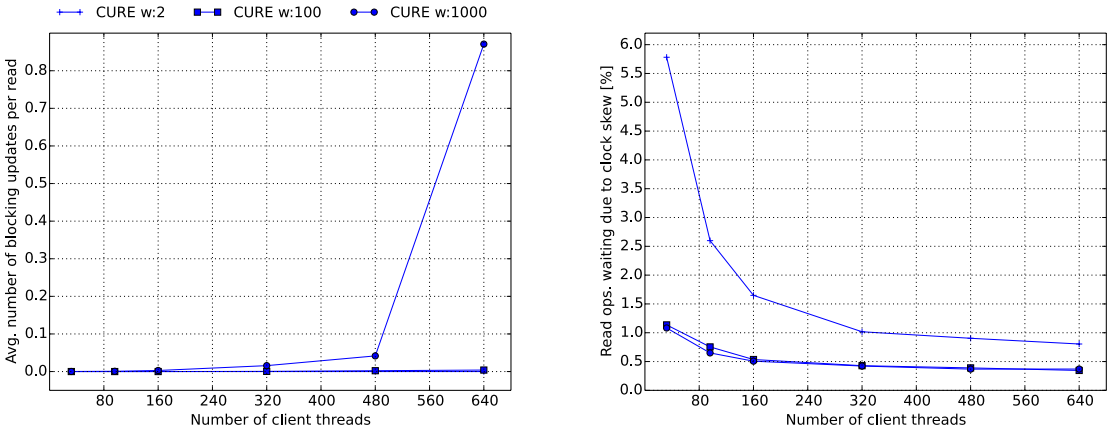
((d)) latency w:2 ((e)) latency w:100 ((f)) latency w:10000

Figure 14.4: Facebook-like read-only transactions (1000 read ops/txn)

However, some effects get diminished while others get augmented. In what follows, we only refer to the differences between results.

Throughput and Latency. Figure 14.4 shows the throughput and latency responses of all protocols under this workload. These transactions exhibit significantly higher —around 10X— latency than single-shot transactions, as they incur 10 rounds of 100 reads each. The latency-throughput trend of all systems is very similar to that of single-shot transactions: CV outperforms the remaining protocols, and the difference becomes larger as update rate augments. One difference with respect to single-shot transactions is that OP exhibits slightly worst performance than the other systems. This happens because of OP's mechanism for enforcing causal order: every time a transaction coordinator receives a read response, it must recompute the transaction's causal dependencies (Algorithm 13.1, Lines 11 and 12). Under this workload, each transaction coordinator performs this computation 1000 times. Nevertheless, the protocol could be modified to avoid this situation by performing such computation in parallel with subsequent read operations, which we have not experimented with.

Cure's Latency. Under this workload, where transactions execute for a long time, the blocking cases of Cure are significantly reduced with respect to those of single-shot transactions. **Figure 14.5(a)** shows the percentage of read operations that blocked due to clock skew under Cure. As we see, the effect practically disappears —below 6% of reads block— under all workloads. If we



((a)) Wait due to committing update

((b)) Wait due to clock skew

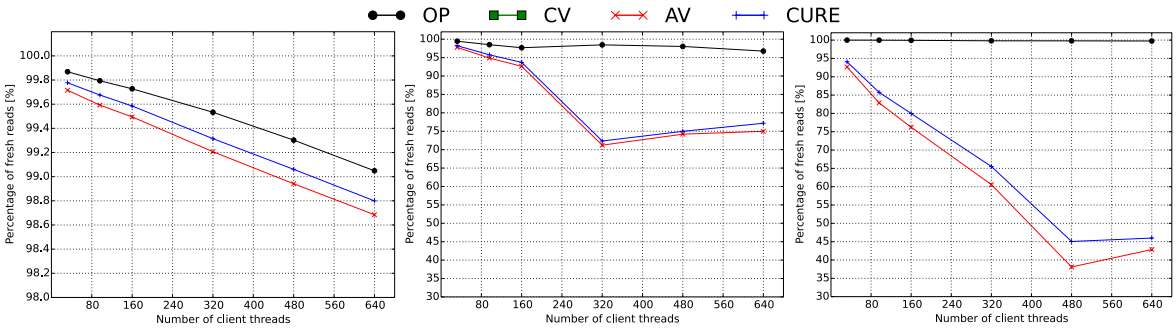
Figure 14.5: Cure blocking scenarios - Facebook-like read-only-transactions (100 read ops/txn)

consider that each read round takes approximately 10ms, rounds after the first one are very unlikely to block due to clock skew, where most of waiting is expected to happen. The same occurs with blocking due waiting for update transactions to commit, as shown in **Figure 14.5(b)**. Under maximum contention, under 1% of read operations block due to this effect.

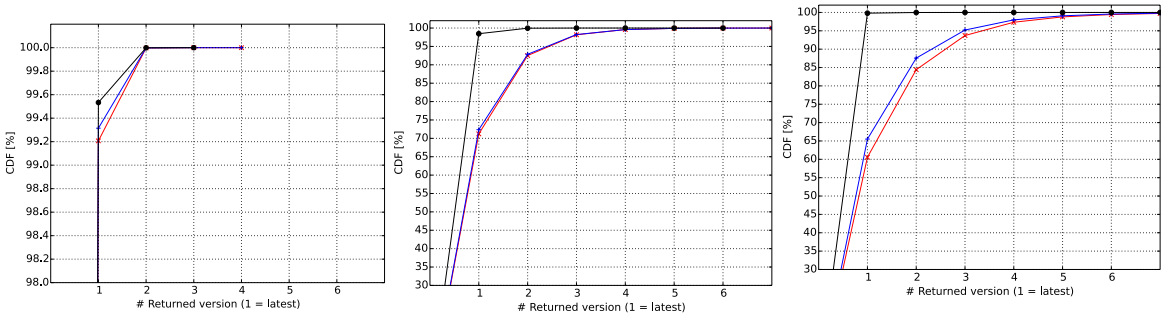
Freshness and multi-version overhead. Figures **14.6(a)**, **14.6(b)** and **14.6(c)** show the freshness response as the number of client threads increases, for different update rates. Figures **14.6(d)**, **14.6(e)** and **14.6(f)** display a CDF showing how stale a read version is under 360 client threads for each update rate.

The trend is similar to that of single-shot read-only transactions (Figure 14.3): OP outperforms the remaining protocols under all configurations, and Cure and AV degrade freshness significantly as contention is added to the system. For all protocols, the effect of staleness gets magnified with respect to that of single-shot transactions. This occurs because transactions are long lived, which renders interleaving with update transactions more frequent. The worst-case scenarios are 5% of stale updates for OP, while 62% for AV, and 55% for Cure. In terms of oldest versions read under contention (50% of updates and maximum client load), OP returned, at most, the 7th-to-latest version, while Cure the 28th and AV the 31st. Figures 14.6(g), 14.6(h), and 14.6(i) show the multi-version-overhead results under this workload. Graphs follow a similar-but-magnified trend too as that of single-shot reads, where overhead peaks at 1.05X for OP, around 2.2X for Cure, and 2.4X for AV.

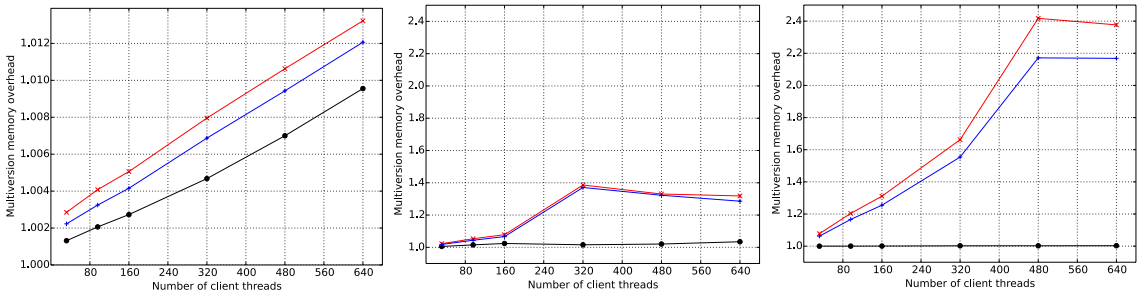
Conclusion. Under this workload, we have observed a different effect of the trade-off over these protocols. First, as transactions live long time, all protocols exhibit similar latency, including Cure, which is not latency optimal. In terms of freshness, we observe that protocols with atomic visibility get highly penalised as contention increases.



((a)) freshness w:2 ((b)) freshness w:100 ((c)) freshness w:10000



((d)) freshness w:2 threads:320 ((e)) freshness w:100 threads:320 ((f)) freshness w:10000 threads:320



((g)) MV overhead w:2 ((h)) MV overhead w:100 ((i)) MV overhead w:1000

Figure 14.6: Freshness of Facebook-like read-only transactions

Chapter 15

Conclusion of Part II

Large-scale cloud services pose strict requirements on their storage. In particular, they require low-latency reads, fact that has motivated designers to move away from transactional guarantees. Under the requirement of no extra delays (with respect to a non-transactional system), strengthening the guarantees of a distributed read algorithm affects the freshness of the data it can read. We explore the three-way trade-off between a transactional read algorithm's isolation guarantees, its delays, and its freshness, and analyse a spectrum of possible points in the design space. We consider three read guarantees. Committed Visibility, the weakest, disallows observing uncommitted data. We identify the intermediate Order-Preserving Visibility, which further disallows observing gaps, given order of updates. Finally, Atomic Visibility is the strongest. It further guarantees a transaction does not read other transaction's updates partially. In summary, we find that minimal-delay reads ensuring Atomic Visibility penalise data freshness significantly, as they force transactions to read updates committed in the past. Minimal-delay Order-Preserving reads offer nearly-optimal freshness, as they allow reading concurrent updates. Moreover, to guarantee reading the most up-to-date data ensuring Order-Preserving (or Atomic) Visibility is only possible by implementing mutually-exclusive reads and updates, where one kind of operation might delay the other indefinitely.

We have used these results to propose new isolation levels that leverage the latency/freshness properties of Order-Preserving reads. TCC$^-$ results from degrading TCC's Atomic reads to Order Preserving. PSI$^-$ results from applying the same modification to Parallel Snapshot Isolation (PSI). These models offer weaker guarantees than the stronger TCC and PSI. On the positive side, they can attain significantly better freshness. In the case of PSI$^-$, this can reduce an implementation's abort rate and, therefore, improve its throughput.

Our results have helped us design new protocols. We have modified Cure, which exhibits blocking delays, to create three minimal-delay protocols: AV maintains its (Atomic) read guarantees and achieves guarantees reading with minimal delays by degrading freshness. The the other two improve freshness in different degrees by degrading read guarantees. OP ensures TCC$^-$. CV ensures Read Committed Isolation, by further degrading Cure's reads to Committed Visibility.

The evaluation of these three protocols supports the theoretical conclusions of the trade-off.

We expect the introduced results and algorithm designs will help the task of designing future systems.

Part III

Related Work

Chapter 16

Causal Consistency for Cloud Deployments

In this section, we analyse how existing cloud storage systems and protocols implement causal consistency with respect to the notions introduced in Section 3.5.2.1 .

16.1 Causally-Consistent Systems

Recently, a number of causally-consistent, partitioned and geo-replicated data stores were proposed. Table 16.1 summarises some of the details of how they achieve causal consistency. In order to decide when remote updates can be made visible, COPS, Eiger, ChainReaction and Orbe use mechanisms that rely on piggybacking causal dependency information with updates and exchanging explicit dependency check messages among partitions at remote data centres. Even when they employ various optimisations to reduce the size of dependencies and the number of messages, in the best case, their metadata grows linearly with the number of partitions. GentleRain avoids such expensive checks. Instead, it uses a global stabilisation algorithm for making updates visible at remote data centres. This algorithm achieves throughput close to eventually consistent systems, at the cost of increased remote update visibility latency. Cure, AV, and OP follow this design choice and achieve similar throughput while providing stronger semantics. Furthermore, by using a vector clock sized with the number of replicas, our protocols are able to reduce remote update visibility latency and increase resiliency to network partitions and data centre failures (Explained in Section 5.3).

Previous systems offer a variety of limited but interesting transactional interfaces that aim at easing the development of applications. COPS [60] introduced the concept of causally-consistent read-only transactions, which other solutions, such as ChainReaction [18], Orbe [45] and GentleRain [47], adopted. Eiger [61], extended this transactional interface with causally-consistent write-only transactions. Cure, AV, and OP provide programmers with general transactions, where each transaction supports multiple rounds of causally-consistent reads, and atomic updates.

System / Property	**Metadata Size**	**Visibility Mechanism**	**Transactional Interface**	**Convergence Mechanism**
COPS [60]	O(objects)	Explicit dep. checks	Read-Only	LWW
Eiger [61]	O(objects)	Explicit dep. checks	Read-Only Update-Only	LWW
ChainReaction [18]	O(DCs)	Explicit dep. checks	Read-Only	LWW
Orbe [45]	O(DCs x partitions)	Explicit dep. checks	Read-Only	LWW
GentleRain [47]	O(1)	Update stabilisation	Read-Only	LWW
Cure	**O(DCs)**	**Update stabilisation**	**General**	**CRDTs**
OP	**O(DCs)**	**Update stabilisation**	**General**	**CRDTs**
AV	**O(DCs)**	**Update stabilisation**	**General**	**CRDTs**

Table 16.1: Property comparison of causally-consistent systems

Previous systems achieve convergence using a last-writer-wins policy [85]. Our systems further provide support for confluent data types (CRDTs), which were introduced in Section 3.5.2.3.

16.1.1 Single-machine Causal Consistency

A number of single-machine-per-replica systems have acknowledged the usefulness and applicability of causal consistency. Early examples that had a profound impact in research are the ISIS toolkit [53], Bayou [72], lazy replication [55], causal memory [14] and PRACTI [26].

Recently, SwiftCloud [89] addressed the problem of providing TCC in a fashion that tolerates full-replica failures. Its design ensures that updates are made visible to readers when they have applied at a number (k) of sites. Finally, Saturn [32] introduced the idea of dependency dissemination trees. It exhibits the same progress properties as our introduced protocols, with reduced metadata. This allows Saturn to provide better scalability under large number of replicas. Saturn is not designed for cloud deployments, where a replica might be partitioned in a large number of servers. In this setting, Saturn exhibits a bottleneck of sending and receiving updates through a single process per replica.

16.2 Strongly-Consistent Systems Enforcing Causal Consistency

Many systems that enforce strong transactional semantics include a mechanism to ensure causal consistency. Table 16.2 summarises the discussion of this section. As these systems totally order

System* / *Property	**Criteria**	**Metadata Size**	**Visibility Mechanism**	**Transactional Interface**
Jessy [75]	NMSI	O(partitions)	Sync. Replication	General
Walter [81]	PSI	O(partitions)	Local Stabilisation	General
Occult [65]	PSI	O(partitions)	No dep. checks	General
Blotter [66]	PSI	O(partitions)	No dep. checks	General

Table 16.2: Property comparison of strongly-consistent systems enforcing causal consistency

updates to each object, they do not allow concurrent conflicting updates and, therefore, they do not require a convergence mechanism. Moreover, all considered systems offer a general transactional interface. Jessy and GMU ensure causal consistency similarly: they use vector clocks sized with the number of partitions to time stamp transactions following causal order, and they propagate updates synchronously across replicas, making updates visible at all replicas simultaneously. Walter and Occult use a similar dependency encoding. Walter's architecture consists of preferred and cache replicas. A preferred replica receives updates to its preferred objects, applies them locally and sends it asynchronously to cache replicas. Walter makes updates visible stabilising them locally, as each replica is not partitioned. Occult is designed with partitioning in mind. This protocol uses no dependency checks, and delegates the task of checking reads are causally consistent to the read protocol.

Chapter 17

Comparison of Transactional Systems

In this section, we discuss the transactional properties of existing systems and research prototypes, and relate them to the trade-off. Table 17.1 summarises the discussion.

17.0.1 Weakly-consistent systems

Systems that are designed for high-availability and low latency under replication are those that do not require a per-object monotonic total order of updates, and thus avoid synchronous replication.

Somewhat surprisingly, even when these systems are designed for low latency, minimal-delay designs with read isolation are missing from the literature, as they all incur bounded delays.

No Isolation. LinkedIn's Espresso [73], Facebook's Tao [33], Yahoo's PNUTS [39], Amazon's Dynamo [43], Twitter's Rainbird [2], and Google's BigTable [37] ensure optimal reads but no atomic updates nor read isolation. Cassandra offers atomic updates and no isolation [42]. Reads are only prevented from observing partial updates within a single row [42]. Its design targets replication, under which it offers a wide spectrum of per-object (called row) consistency guarantees. These range from no ordering whatsoever to monotonic total order, obtaining per-object linearisability when combined with strong reads. This system does not provide any across-object update ordering [41].

Committed Visibility. MySQL cluster provides Committed Visibility by locking; reads sometimes block waiting for a transaction to commit [9]. This work introduces CV, a lightweight protocol with atomic updates that implements Committed Visibility with optimal reads.

Order-Preserving Visibility. Two previous systems implement Order-Preserving Visibility in the literature. COPS [60] incurs multiple read-rounds. COPS-SNOW [63] guarantees order-preserving visibility with minimal delay and concurrent freshness. When compared to the original COPS, it removes the second round of reads by rendering updates expensive —an update

System* / *Property	**Atomic Upd.**	**Upd. Order**	**Model**	**Delay**	**Fresh.**	**Read Isolation**	**Sector**
PNUTS [39]	-	-	-	Minimal	Latest	-	-
Dynamo [43]	-	-	-	Minimal	Latest	-	-
Rainbird [2]	-	-	-	Minimal	Latest	-	-
BigTable [37]	-	-	-	Minimal	Latest	-	-
Espresso [73]	-	-	-	Minimal	Latest	-	-
Tao [33]	-	-	-	Minimal	Latest	-	-
Cassandra [42]	✓	-	-	Minimal	Latest	-	-
MySQL Cluster [9]	✓	-	RC	Bounded	Latest	CV	3
CV	✓	-	RC	Minimal	Latest	CV	3
COPS [60]	-	Causal	CC	Bounded	Conc.	OP	/5
COPS-SNOW [63]	-	Causal	CC	Minimal	Conc.	OP	2
OP	✓	Causal	**TCC**$^-$	Minimal	Conc.	OP	2
GentleRain [47]	-	Causal	CC	Bounded	Conc.	AV	5
Orbe [45]	-	Causal	CC	Bounded	Conc.	AV	5
ChainReaction [18]	-	Causal	CC	Bounded	Conc.	AV	5
Cure	✓	Causal	TCC	Bounded	Conc.	AV	5
Eiger [61]	✓	Causal	TCC	Bounded	Conc.	AV	5
RAMP [24]	✓	Causal	RA	Bounded	Conc.	AV	5
AV	✓	Causal	TCC	Minimal	Stable	AV	1
Jessy [75]	✓	Causal	NMSI	Bounded	Conc.	AV	5
Walter [81]	✓	Causal	PSI	Bounded	Conc.	AV	5
Occult [65]	✓	Causal	PSI	Mutex R/W	Conc.	AV	5
Blotter [66]	✓	Causal	NMSI	Bounded	Conc.	AV	5
GMU [71]	✓	Causal	US	Bounded	Conc.	AV	5
Clock-SI [46]	✓	Monotonic	SI	Bounded	Conc.	AV	5
CockroachDB-SI [38]	✓	Monotonic	SI	Bounded	Conc.	AV	5
CockroachDB-S [38]	✓	Monotonic	S	Bounded	Conc.	AV	5
Spanner ROTX [40]	✓	Monotonic	SS	Minimal	Stable	AV	1
Spanner [40]	✓	Monotonic	S	Mutex R/W	Latest	AV	9
Rococo [67]	✓	Monotonic	SS	Mutex R/W	Latest	AV	9
Rococo-SNOW [63]	✓	Monotonic	SS	Mutex R/W	Latest	AV	9

Table 17.1: Guarantees, delay and freshness for several published systems. The sector numbers cross-reference to Figure A.1; /5 refers to the second plane of Sector 5.

operation must update data structures of all the objects it causally depends on. Both of these systems rely on metadata sized with the number of objects in the system to causally-order updates. None of them support atomic updates.

The introduced OP has the same read semantics without incurring such costs, and furthermore providing atomic updates.

Atomic Visibility. Many weakly consistent systems require blocking reads to achieve Atomic Visibility. GentleRain [47], Orbe [45], ChainReaction [18], and Cure [16] block for a short time to wait for concurrent transactions to commit. Moreover, Cure, Orbe and GentleRain also block temporarily in the case of clock skew between servers. GentleRain, Orbe, and ChainReaction do

not support all-or-nothing updates. Interestingly, if they would, then their snapshot algorithm would guarantee Atomic Visibility with no modification. The remaining considered systems incur multiple rounds of reads. Examples include Eiger [61], and RAMP —which establishes a per-object partial order of updates [24]. This work introduced AV, the first weakly-consistent protocol that achieves minimal delays and Atomic Visibility. It implements Transactional Causal Consistency.

17.0.2 Strongly-consistent systems

Atomic Visibility. With the exception of (per-object) strongly-consistent modes of Cassandra —those that require performing updates synchronously across data centres [41]— which provides no read isolation, all strongly consistent systems implement Atomic Visibility.

Walter, Occult, Blotter, Jessy and GMU partially order updates according to causal order and enforce a monotonic total order of the updates of each object (avoiding conflicting writes). They achieve concurrent freshness with bounded delays: Walter retries reads, Blotter, Jessy and GMU read sequentially. In Occult reads speculatively attempt to read from an atomic snapshot from sites that might be in an inconsistent state; it exhibits unbounded delays, as it aborts read-only transactions when an inconsistent read is detected.

Clock-SI provides a total monotonic order of updates —which is required by Snapshot Isolation. Its read algorithm is very similar to that of Cure and GentleRain; it exhibits bounded delays, as it blocks in the case of clock skew or waiting for transactions to commit.

CockRoachDB offers Snapshot Isolation and Serialisability, both ensuring Atomic Visibility. Under Snapshot Isolation, reads might block waiting for a transaction to commit. Under Serialisability, read-only transactions might abort when they detect a serialisation conflict.

Spanner features two kinds of transactions. Strictly serialisable transactions require latest freshness and atomic visibility. This is enforced by locking, which ensures mutually-exclusive reads and writes (by Proposition 9, this is unavoidable). Spanner's serialisable read-only transactions weaken freshness to stable, and achieve minimal delays.

Rococo guarantees Strict Serialisability, which requires latest freshness and Atomic Visibility. Unavoidably, it resorts to mutually-exclusive reads and writes. Its read algorithm issues an unbounded number of rounds to ensure its desirable guarantees. Rococo-SNOW behaves similarly, with the difference of issuing a bounded number of read rounds, and blocking updates when these rounds do not attain Atomic Visibility and latest freshness. Once updates are stopped, this is guaranteed.

Conclusions and Future Work

Chapter 18

Conclusion

Today's internet-scale services pose strict requirements on cloud storage systems. These systems must handle a large number of requests from users worldwide, offer fast response, remain available in the presence of failures, and provide meaningful guarantees to programmers. High latencies and downtimes directly affect revenues, as they determine the way users interact with a service. The lack of meaningful guarantees complicates the task of programming the application logic. In this thesis, we have studied the problem of building storage providing meaningful guarantees that do not penalise response times and availability.

To reduce latencies, maximise availability and handle large load, cloud services exhibit a widely distributed architecture consisting of multiple data centres. A data centre can handle load beyond what a single machine can handle. A user can minimise latency by connecting to his closest site. In the presence of network failures, he can be redirected to another data centre. Storage systems replicate data at each data centre, where data is further scattered across servers. These systems must, therefore, implement communication protocols to access data scattered across servers in a data centre and keeping data centres up-to date.

A transaction is a powerful abstraction that allows to group multiple operations into an atomic unit with well defined guarantees. In particular, strong isolation gives a programmer the illusion of a sequential data store, where a transaction runs alone and does not interfere with the operations of other transactions. The lack of strong isolation results in added more complex application-logic development. As storage exposes anomalous behaviour, this may need to be handled at the application level.

Ensuring strong isolation in a cloud architecture requires cross-data-centre coordination, which affects latency in the normal case, and availability when data centres cannot communicate. In this work, we have studied the design space of weak transactional isolation, which does not require cross-site synchronous communication.

18.1 Contributions

18.1.1 Part I

Our first contribution was the design and implementation of Cure. Cure is a transactional protocol for multi-data centre deployments that provides *(i)* Transactional Causal Consistency (TCC): causal+ consistency, ensuring that if one update happens before another, they will be observed in the same order, *(ii)* support for operation-based replicated data types (CRDTs) such as counters, sets, tables and sequences, with intuitive semantics and guaranteed convergence in the presence of concurrent conflicting updates and partial failures, and *(iii)* general transactions, ensuring that multiple keys (objects) are both read and written consistently. This combination equips Cure with the strongest semantics ever provided by an always-available data store. Cure's design is based on a novel approach to support parallelism between servers within the data centre that minimises the overhead of causal consistency in inter-DC traffic. It makes updates that are known to be safe visible in batches. Cure encodes causal-order metadata as a single scalar per DC —thus incurring small overhead— to improve freshness and resilience to network partitions over previous work.

Our experimental evaluation shows that Cure outperforms state-of-art systems with similar guarantees, and exhibits performance similar to a baseline system offering weak semantics.

18.1.2 Part II

The transactional read algorithms of systems like Cure exhibit latency overheads that have impeded their adoption at scale. Our second contribution was to explore how to implement distributed isolation with no extra delays with respect to a non-transactional system. We formalised the three-way tension between read guarantees, read delay (and hence latency), and freshness, and showed the desirable points of the design space which are possible/impossible.

For the analysis, we have identified a read-isolation property called Order-Preserving Visibility. Order-Preserving reads are weaker than Atomic reads, guaranteed by TCC and stronger models (e.g., Snapshot Isolation and Serialisability). When compared to Atomic Visibility, they do not forbid a concurrency anomaly called Read Skew, which allows observing the updates of other transactions partially. On the positive side, (like Atomic reads) Order-Preserving Visibility disallows reading uncommitted data and observing (e.g. causal) ordering anomalies.

The three-way trade-off between read isolation, delay (latency), and data freshness can be summarised as follows: *(i)* To guarantee reading data that is the most fresh without delay is possible only under a weakly-isolated mode, similar to that provided by the standard Read Committed. *(ii)* Conversely, reads that enforce stronger isolation at minimal delay impose reading data from the past (not fresh). *(iii)* Minimal-delay Atomic reads force transactions to read updates that were committed and acknowledged in the past. *(iv)* On the contrary, minimal-delay Order-Preserving reads can read the updates of concurrent transactions. *(v)* Order-Preserving

and Atomic reads at maximal freshness require mutually-exclusive read and write operations, which may block reads or writes indefinitely. These results hold independently of other features, such as update semantics (e.g., monotonically ordered or not) or data model (e.g., structured or unstructured).

Motivated by these results, we have proposed two isolation properties: TCC$^-$ and PSI$^-$. They result from degrading the (Atomic) read guarantees of TCC and PSI to Order Preserving. Using the results of the trade-off, we have used Cure, which's read algorithm sometimes blocks, to create three protocols with no delays. AV maintains Cure's TCC guarantees by degrading freshness. The other two improve freshness by weakening the isolation guarantees: OP provides TCC$^-$, and CV provides Read Committed Isolation, where reads enforce Committed Visibility.

Experimentally, as expected, the three protocols exhibited similar latency. CV always observed the most recent data, whereas freshness degraded negligibly for OP, and the degradation was severe under Cure and AV.

Chapter 19

Future Work

In this work, we have explored the design space of highly-available transactions.

Protocol design. In Part I, we have presented the design of Cure and three protocols that result from applying different modifications to it. Our algorithms ensuring causal consistency rely on a stabilisation protocol that involves all the partitions of a data centre to make updates visible. If a partition is down or unreachable, other partitions will cease to make remote updates visible. To implement stabilisation protocols in production environments, future work should study fault tolerance mechanisms for partitions, for instance, through replication.

Trade-off analysis. In Part II, we studied the trade-offs of building transactional read algorithms. The algorithms we have compared show interesting points of the design space, but do not explore it completely. We have evaluated the freshness and latency of Cure, which exhibits delays sourced in blocking. It would be interesting to explore the behaviour of systems that rely on retrying operations to ensure read isolation. We have seen that freshness-optimal isolation requires mutually-exclusive reads and writes, which may delay operations indefinitely. It would be interesting to see how freshness-optimal designs affect latency under different read-isolation levels. Also, it would be interesting to explore the effects of the trade-off in strong consistency.

New Isolation Models. In Chapter 12, we have suggested new isolation properties that we have not studied in depth. It would be interesting to analyse what kinds of applications would benefit from these models. We have proposed a model called PSI^{-}, which results from degrading the read guarantees of Parallel Snapshot Isolation (PSI). It would be interesting to implement a transactional algorithm that implements this model and evaluate them empirically.

Related research directions. This work is part of research that aims at simplifying the development of applications over highly-parallel architectures. In particular, we have explored the relationship between semantics and performance from an algorithmic perspective. Other

directions within this field include the development of programming models and tools to help programmers decide and express what model suffices to ensure an application can run safely — unaffected by its exposed anomalies— without sacrificing throughput, latency and data freshness more than what is strictly required.

Part IV

Appendix

Appendix A

Résumée de la Thèse

Les services web grande échelle reposent sur des déploiements fortement distribués et hautement parallèle, afin de supporter de fortes charges et des larges volumes de données. Par exemple, "*Amazon gère une plateforme de commerce en ligne responsable de dizaines de millions de clients en heure de pointe, à travers de dizaines de milliers de serveurs situés dans de multiples data centers dans le monde* [43]", et Tao, la base de données qui stocke le graphe social de Facebook "*tourne sur des milliers de machines géo-distribuées, offrant l'accès à des petabytes de données, et peut supporter un milliard de lectures et des millions d'écritures chaque seconde* [33]".

Ces services doivent répondre aux requêtes avec célérité et offrir une expérience toujours online. Les temps de réponse affectent directement les revenus [44, 76] et, comme reporté par Amazon, "*même la panne la plus anodine des services a des impacts significatifs sur les revenus et la confiance des clients* [43]". Afin de réduire les temps de réponse et tolérer les fautes, ils emploient la géo-réplication: en déployant des répliques de la logique applicative et de son état sur de multiples data centers dans le monde. Les utilisateurs peuvent minimiser la latence en se connectant au réplica le plus proche et, en présence de fautes qui rendraient un data center hors service, peuvent se connecter à un autre data center resté en ligne. Sur chaque data center (ou réplica), l'application est déployée sur de multiples serveurs front-end, et l'état de l'application est partitionné à travers une multitude de serveurs de stockage. Ainsi, chaque réplica peut servir un volume de requêtes et stocker un quantité de données bien au delà de ce que pourrait supporter une seule machine.

Il est bien connu que, dans ce genre de scénario, le design d'un système doit choisir entre la simplicité du développement d'une application, ou sa disponibilité et réactivité:

- En présence de connexions longue distance et inter-continentales, les partitions (P) du réseau sont inévitables.

  Selon le théorème de CAP [50], un système géo-distribué doit faire le choix entre haute disponibilité (A) ou cohérence forte (C); garantir ces deux propriétés simultanément est impossible.

Le choix d'une cohérence forte simplifie le développement de la logique applicative, car il dissimule la complexité de la géo-réplication en gardant les data-centers synchronisés continuellement. Néanmoins, il expose les utilisateurs à de fortes latences et aux pannes des liens réseau étendu. Au contraire, un autre design choisirait de favoriser la réactivité et la disponibilité en répondant aux requêtes utilisateur depuis le réplica le plus proche, en évitant les inconvénients des communications entre data centers, et en synchronisant ceux-ci de manière paresseuse [55], cependant ceci expose la concurrence, ce qui rend le développement de la logique applicative plus complexe et sujette à erreur [25].

- Lire et modifier des données éparpillées sur de multiples machines de manière cohérente demande d'implémenter des transactions distribuées qui respectent l'atomicité, la propriété du tout ou rien des mises à jour, et l'isolation des lectures, qui garantit, par exemple, que toutes les mises à jour créées atomiquement seront observées simultanément[24] et l'absence d'incohérences liées à l'ordre des opérations [60] ainsi que d'autres anomalies liées à la concurrence[19]. Les transactions distribuées peuvent dissimuler à la logique applicative la complexité du caractère distribué de l'application, mais entraînent souvent des surcharges de communication et des scénarios bloquants qui impactent directement la latence. [15, 63].

Les architectes d'applications aux besoins dominés par la latence on été amenés, pour ces raisons, à abandoner la cohérence et à adopter des opérations sans garanties transactionnelles mais plus rapides, exposant ainsi les développeurs d'applications, et parfois les utilisateurs à des anomalies liées à la distribution et la réplication. Par exemple Tao de Facebook[33], Espresso de LinkedIn [73], PNUTS de Yahoo [39] et Dynamo d'Amazon[43].

Dans cette thèse, nous avons étudié la construction de bases de données distribuées et géo-répliquées qui offrent des sémantiques transactionnelles tout en garantissant une disponibilité et une réactivité similaire à celles de systèmes qui, comme vu ci-dessus, n'offrent aucune garantie transactionnelle ou de cohérence. Nous commencerons par présenter Cure[1], un protocole transactionnel qui offre une sémantique simple et qui reste compatible avec une haute disponibilité et des latences faibles. Cure implémente la Cohérence Causale Transactionnelle (TCC: Transactional Causal Consistency) et supporte des types de données répliquées sans conflits (CRDTs: Conflict-free Replicated Data Types). Cure offre ces garanties tout en atteignant des performances similaires à celles de sémantiques plus faibles. Dans un second temps, nous analyserons comment construire des protocoles de transactions distribuées qui ne souffrent pas de délais supplémentaires comparés à des systèmes non transactionnels. Nous démontrerons un compromis entre l'isolation des lectures, les latences et la fraîcheur des données (l'âge des valeurs lues par la transaction). Nous mettrons à profit les résultats pour modifier Cure, qui souffre de

[1] Cure est le noyau transactionnel qui offre les garanties fondamentales de la base de données Antidote [4], un projet destiné à fournir aux applications un stockage cohérent qui ne souffre que des synchronisations minimales pour assurer leurs invariants. Durant cette thèse, j'ai été un collaborateur actif au développement d'Antidote.

scénarios bloquants, afin d'en tirer de nouveaux niveaux d'isolation et des protocoles sans délais supplémentaires.

A.0.1 Part I - Cure: Des sémantiques fortes liées à une haute disponibilité et des latences faibles

Afin de soulager le compromis entre la facilité du développement et les performances, des travaux récents se sont concentrés sur l'amélioration des designs AP qui offrent des sémantiques plus fortes [60, 61, 81]. Cure est notre contribution dans cette direction. Tout en offrant de la disponibilité et des performances, Cure offre *(i)* de la Transactional Causal Consistency (TCC), c-à-d la cohérence causale, garantissant que si une opération se produit avant une autre, elles seront observées dans le même ordre, *(ii)* le support pour les types de données répliqués (CRDTs) basés sur des opérations telles que: les compteurs, les ensembles, les tableaux et les suites, avec des sémantiques intuitives, et une convergence garantie même en présence de mises à jour concurrentes ou de pannes partielles, et *(iii)* des transactions générales, qui assurent que de multiples clefs (objets) sont lues et écrites de manière cohérente.

La Cohérence Causale (CC) présente un juste équilibre dans le compromis entre la cohérence et la disponibilité [14, 60]. Il s'agit du modèle de cohérence le plus fort qui soit compatible avec la disponibilité pour des opérations sur des objets uniques[20]. Et puisqu'elle assure la cohérence causale (présentée en section 3.5), elle facilite la réflexion pour les développeurs et pour les utilisateurs. Considérons une utilisatrice qui poste une photo sur son profil, puis poste un commentaire sur cette même photo. Sans la cohérence causale, un autre utilisateur pourrait voir le commentaire mais ne pas pouvoir voir la photographie. Éviter ce type d'anomalie requiert des efforts supplémentaires de programmation au niveau de la logique applicative.

Les CRTDs sont des types de données de haut niveau, dont l'utilisation est aisée pour les développeurs et qui présentent une sémantique riche (Section 3.5.2.3). Les opérations sur des CRDTs ne sont pas seulement des opérations équivalentes à l'usage de registres, mais des méthodes correspondant au type de l'objet CRDT utilisé. Les CRDTs assurent la convergence éventuelle de l'ensemble des réplicas, malgré des mises à jour simultanées et conflictuelles. Des systèmes antérieurs en cohérence causale+ [18, 45, 47, 60, 61] offraient cette garantie de convergence à travers un mécanisme de *dernier écrivain gagne* (LWW: Last Writer Wins), où la mise à jour qui s'est produite le plus récemment prime et écrase les précédentes. Cure offre le support pour des CRDTs basés sur les opérations. Par exemple, les développeurs de Bet365 rapportent qu'utiliser les CRDTs *Set* a changé leur vie, les libérant de détails de bas niveaux et du besoin de corriger les anomalies de concurrence[64].

Opérer de multiples opérations dans une même transaction permets à l'application de maintenir des rapports entre de multiples objets. *L'isolation AP* rejette les propriétés traditionnelles de l'isolation forte, pour lesquelles une synchronisation est nécessaire, en faveur de la disponibilité et de latences faibles [22, 35]. Des systèmes antèrieurs qui implémentent CC+ offrent soit une

lecture depuis un snapshot [18, 45, 47, 60, 61] soit l'atomicité des mises à jour [24, 61]; là où les transactions de Cure offrent les deux.

Prises ensembles, ces propriétés offrent aux développeurs des sémantiques qui sont à la fois claires et fortes. De fait, puisque Cure repose sur une combinaison de ces trois propriétés, il offre les sémantiques les plus fortes qu'une base de données hautement disponible n'ait jamais offerte.

Le design de Cure se base sur une approche novatrice qui facilite le parallélisme entre serveurs au sein d'un data center en minimisant les surcharges de communication liées à la cohérence causale [47]. Contrairement à l'approche usuelle qui consiste à vérifier si une mise à jour reçue satisfait les conditions de la causalité, ce qui demande d'attendre une réponse de la part du serveur distant —que l'on appelle explicitement un message de contrôle de dépendance— Cure s'appuie sur la stabilisation des dépendance, qui rend les mises à jour visibles par lots parmi celles pour qui on a l'assurance que toutes leurs dépendances sont satisfaites. L'apport de Cure comparé aux travaux qui le précèdent est l'encodage des méta données d'ordre causal au sein d'un simple scalaire par data center — ce qui minimise les surcharges — afin d'améliorer la fraîcheur et la résilience aux partitions du réseau comparé aux mécanismes présents dans l'état de l'art (See Section 6.5).

En résumé, les contributions de cette partie de mes travaux sont les suivantes:

- Un nouveau modèle de programmation qui fournit des transactions interactives, causalement cohérentes avec des types de données sans conflit de haut niveau (Chapter 5.1).
- Un protocole haute performance, qui supporte ce modèle de programmation pour des bases de données géo-répliquées (Chapitre 6).
- Une évaluation exhaustive, qui compare notre approche à celles des bases de données de l'état de l'art (Chapitre 7).

A.0.2 Partie II - Le compromis à trois niveaux pour des lectures transactionnelles

Dans cette partie de la thèse, nous étudions les coûts de lecture de données dans une base de donnée transactionnelle distribuée. En particulier, nous tentons de comprendre s'il est possible d'offrir de fortes garanties sur les lectures tout en assurant leur rapidité et la fraîcheur de leur résultats. Il est bien connu que de fortes garanties sont accompagnées par des coûts plus élevés: les algorithmes fonctionnant sur les verrous, les tentatives multiples ou la lecture de données anciennes afin d'isoler les transactions. Au contraire, certains systèmes rejettent complètement cette isolation afin d'en éviter les coûts: un article récent de Facebook (dont la performance est fortement liée aux lectures de données) déclare: *" des propriétés plus fortes ont le potentiel d'améliorer l'expérience utilisateur et de simplifier le développement de la logique applicative [... mais] reposent sur des communications supplémentaires et des mécanismes de gestion d'état plus lourds, ce qui augmente les latences [...] Ce qui pourrait amener à un déclin de l'expérience*

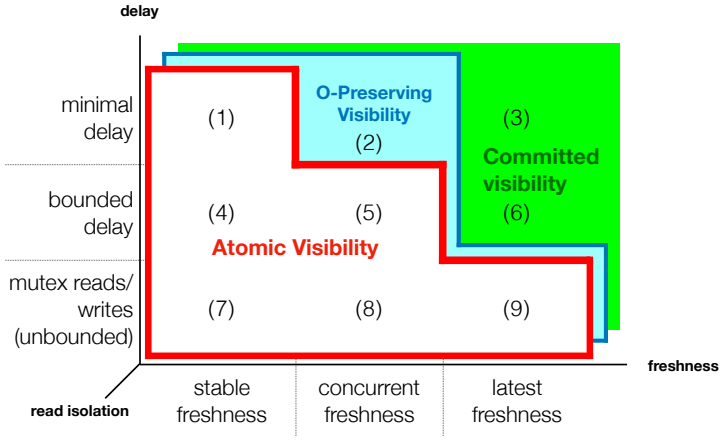


Figure A.1: Le compromis à trois niveaux. Les aires présentent les combinaisons possibles de garanties/délais de lectures/fraîcheur. On trouve les meilleures performances en haut à droite; les garanties sont plus fortes au premier plan. Les numéros de secteurs sont référencés dans la Table 17.1.

utilisateur, et potentiellement à un préjudice " [15]. Cette méfiance est-elle justifiée, c-à-d, des lectures rapides sont elles de manière inhérente impossibles à combiner avec de fortes garanties, ou peut-on améliorer la situation à travers de meilleures isolations? Ces travaux offrent une étude formelle et opérationnelle de ces coûts et des ces compromis. Nous formalisons la tension entre les impératifs des garanties des lectures, des délais de lecture (et donc les latences), et de la fraîcheur, et montrons les points bénéfiques de l'espace des solutions possibles ou impossibles.

Puisque les garanties non sérialisables (présentées en Section 3.3) peuvent améliorer les performances et la disponibilité, dans ces travaux, nous ne partons pas du principe que les mises à jour sont totalement ordonnées. [1] De plus, nous permettons un affaiblissement de l'isolation des lectures: en addition à la visibilité atomique, la garantie la plus forte et qui est présupposée par les modèle transactionnels classiques, nous identifions la visibilité conservatrice d'ordre (Order-Preserving Visibility), elle assure l'absence d'anomalies d'ordres mais tolère les *read skew* (Section 3.3.3) , et nous examinons *Committed Visibility*, qui offre des garanties de lectures équivalentes à l'isolation Read Committed (Section 3.3.5.3). Finalement, nous considérons aussi la dimension de la *fraîcheur*, car (comme nous le montrons) la diminution des délais de lecture peut parfois nous obliger à lire une version des données qui n'est pas la plus récente.

Le schéma A.1 illustre le compromis à trois niveaux entre les garanties, les délais, et la fraîcheur des lectures transactionnelles. Par exemple, sous les conditions de visibilité *Order Preserving* ou Atomique, il est possible de lire sans délai additionnel (par rapport à un système non transactionnel), mais dans ce cas les données les plus récentes ne sont pas accessibles. Cependant, nous montrons que les lectures *Order Preserving* avec un niveau de latence minimale permettent

[1] Dans un système répliqué, imposer un ordre total et monotone aux mises à jour permets une cohérence forte (Strong Consistency) sous partition (CP) ; mais, à l'inverse, il est nécessaire d'accepter les mises à jour concurrentes pour offrir la disponibilité sous partition (AP) [79].

d'observer les mises a jour de transactions qui se sont achevées simultanément (Secteur 2). Comme nous le verrons dans notre évaluation, ceci permets une amélioration significative de la fraîcheur par rapport à la visibilité atomique, qui nous force à ne lire que des données qui étaient stables (écrites et confirmées) avant le début de la transaction (Secteur 1). Si, par contre, l'application requiert les données les plus récentes, une visibilité atomique ou *Order Preserving* n'est possible qu'avec un protocole où les lectures et écritures sont mutuellement exclusives, c-à-d qu'une lecture peut être retardée (bloquée ou dans une boucle de nouvelles tentatives) infiniment par des lectures, ou vice-versa (Secteur 9). Finalement, Committed Visibility permets aux transactions d'accéder aux données les plus fraîches sans délai supplémentaire (Secteur 3).

Ce document inclut les contributions suivantes:

1. Une étude formelle des compromis entes les garanties de lecture, les délais, et la fraîcheur des lectures transactionnelles. Nous prouvons quelles combinaisons souhaitables sont possibles, et lesquelles ne le sont pas.
2. Deux nouvelles propriétés d'isolation, TCC- et PSI-, qui résultent de la dépréciation des garanties de lecture de TCC et de PSI, en allant de la visibilité atomique à la Préservation de l'Ordre (causal), ce qui positionne ces modèles différemment par rapport à notre compromis.
3. Trois protocoles à latences minimales dérivés de Cure, qui garantissent la visibilité atomique, la fraîcheur des lectures transactionnelles simultanées et la vivacité des scénarios bloquants (Secteur 5), en nous inspirant des résultats de nos analyses: AV s'assure de TCC dans le Secteur 1, OP s'assure de TCC- dans le Secteur 2, et CV s'assure de la *Read Committed Isolation* dans le Secteur 3. Nous proposons des protocoles au design détaillé, y-compris le pseudo-code. À notre connaissance, ces protocoles sont les premiers à offrir ces garanties d'une manière qui assure des délais minimaux.
4. Une évaluation de ces protocoles pour valider nos résultats de manière empirique. Durant nos expériences, nous comparons les protocoles présentés et nous les comparons à Cure. Nos protocoles à délais minimaux manifestent des latences similaires. CV observe les données les plus récentes là où la fraîcheur est dégradée de manière négligeable pour OP, et de manière sévère pour AV.

Nous espérons que ces résultats aideront les architectes de systèmes distribués dans leur prise de décision lorsqu'ils devront séléctionner ou construire des stockages transactionnels.

Bibliography

[1] Basho bench. http://github.com/SyncFree/basho_bench.

[2] Rainbird: Real-time analytics@ twitter. https://www.slideshare.net/kevinweil/rainbird-realtime-analytics-at-twitter-strata-2011.

[3] Twitter, inc. https://twitter.com/.

[4] AntidoteDB. http://syncfree.github.io/antidote/, 2015.

[5] NTP: The network time protocol. https://www.ntp.org/, retrieved August 2015.

[6] ZeroMQ. http://http://zeromq.org/, 2015.

[7] NoSQL Engagement Database | Couchbase, 2018. URL https://www.couchbase.com/.

[8] The outrageous costs of data center downtime, 2018. URL https://html.com/blog/data-center-downtime/#ixzz53aaj5PIt.

[9] MySQL :: MySQL NDB Cluster :: Limits Relating to Transaction Handling in NDB Cluster. https://dev.mysql.com/doc/mysql-cluster-excerpt/5.7/en/mysql-cluster-limitations-transactions.html, 2018.

[10] Couchbase, Run Your First N1QL Query, 2018. URL https://developer.couchbase.com/documentation/server/5.0/getting-started/try-a-query.html.

[11] The new york times | stock traders find speed pays, in milliseconds, 2018. URL https://nyti.ms/2k2Kncl.

[12] Marcos K. Aguilera, Joshua B. Leners, and Michael Walfish. Yesquel: Scalable SQL Storage for Web Applications. SOSP '15, pages 245–262, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3834-9. doi: 10.1145/2815400.2815413. URL http://doi.acm.org/10.1145/2815400.2815413.

[13] Mustaque Ahamad, James E. Burns, Phillip W. Hutto, and Gil Neiger. Causal memory. In *Proc. 5th Int. Workshop on Distributed Algorithms*, pages 9–30, Delphi, Greece, October 1991.

[14] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, March 1995. doi: 10.1007/BF01784241. URL http://dx.doi.org/10.1007/BF01784241.

[15] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. Challenges to Adopting Stronger Consistency at Scale. In *HOTOS*, pages 13–13, Berkeley, CA, USA, 2015. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2831090.2831103.

[16] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, pages 405–414, Nara, Japan, June 2016. doi: 10.1109/ICDCS.2016.98. URL http://doi.ieeecomputersociety.org/10.1109/ICDCS.2016.98.

[17] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient state-based CRDTs by delta-mutation. In *Int. Conf. on Networked Systems (NETYS)*, volume 9466 of *Lecture Notes in Comp. Sc.*, pages 62–76, Agadir, Morocco, May 2015. Springer-Verlag. doi: 10.1007/978-3-319-26850-7_5. URL http://dx.doi.org/10.1007/978-3-319-26850-7_5.

[18] Sérgio Almeida, João Leitão, and Luís Rodrigues. ChainReaction: A causal+ consistent datastore based on chain replication. In *Euro. Conf. on Comp. Sys. (EuroSys)*, pages 85–98, Prague, Czech Republic, 2013. doi: 10.1145/2465351.2465361. URL http://doi.acm.org/10.1145/2465351.2465361.

[19] ANSI. X3. 135-1992, American National Standard for Information Systems-Database Language-SQL, 1992.

[20] Hagit Attiya, Faith Ellen, and Adam Morrison. Limitations of highly-available eventually-consistent data stores. *IEEE Trans. on Parallel and Dist. Sys. (TPDS)*, 28(1):141–155, January 2017. doi: 10.1109/TPDS.2016.2556669. URL https://doi.org/10.1109/TPDS.2016.2556669.

[21] Peter Bailis and Kyle Kingsbury. The network is reliable: An informal survey of real-world communications failures. *ACM Queue*, 2014.

[22] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *Proc. VLDB Endow.*, 7(3):181–192, November 2013. doi: 10.14778/2732232.2732237. URL http://dx.doi.org/10.14778/2732232.2732237.

[23] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Hat, not cap: Towards highly available transactions. In *Proceedings of the 14th USENIX Conference on*

Hot Topics in Operating Systems, HotOS'13, pages 24–24, Berkeley, CA, USA, 2013. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2490483.2490507.

[24] Peter Bailis, Alan Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. Scalable Atomic Visibility with RAMP Transactions. In *SIGMOD*, pages 27–38, New York, NY, USA, 2014. ACM. doi: 10.1145/2588555.2588562. URL http://doi.acm.org/10.1145/2588555.2588562.

[25] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1327–1342, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2737784. URL http://doi.acm.org/10.1145/2723372.2737784.

[26] N Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Networked Sys. Design and Implem. (NSDI)*, pages 59–72, San Jose, CA, USA, May 2006. Usenix, Usenix. URL https://www.usenix.org/legacy/event/nsdi06/tech/belaramani.html.

[27] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Rec.*, 24(2):1–10, May 1995. ISSN 0163-5808. doi: 10.1145/568271.223785. URL http://doi.acm.org/10.1145/568271.223785.

[28] Philip A. Bernstein and Nathan Goodman. Multiversion Concurrency Control; Theory and Algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, December 1983. ISSN 0362-5915. doi: 10.1145/319996.319998. URL http://doi.acm.org/10.1145/319996.319998.

[29] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987. ISBN 0-201-10715-5.

[30] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set. Rapport de Recherche RR-8083, Institut National de la Recherche en Informatique et Automatique (Inria), Rocquencourt, France, October 2012. URL http://hal.inria.fr/hal-00738680.

[31] Anna Bouch, Allan Kuchinsky, and Nina Bhatti. Quality is in the eye of the beholder: meeting users' requirements for internet quality of service. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 297–304. ACM, 2000.

[32] Bravo, Manuel and Rodrigues, Luís and Van Roy, Peter. Saturn: A Distributed Metadata Service for Causal Consistency. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 111–126, New York, NY, USA, 2017. ACM. ISBN

978-1-4503-4938-3. doi: 10.1145/3064176.3064210. URL http://doi.acm.org/10.1145/3064176.3064210.

[33] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *USENIX ATC*, pages 49–60, San Jose, CA, 2013. USENIX. ISBN 978-1-931971-01-0. URL https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson.

[34] Protocol Buffers. Google's data interchange format, 2011.

[35] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Mooly Sagiv. Eventually consistent transactions. In *Euro. Symp. on Programming (ESOP)*, Tallinn, Estonia, March 2012. doi: http://dx.doi.org/10.1007/978-3-642-28869-2_4.

[36] Irina Ceaparu, Jonathan Lazar, Katie Bessiere, John Robinson, and Ben Shneiderman. Determining causes and severity of end-user frustration. *International journal of human-computer interaction*, 17(3):333–356, 2004.

[37] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008. ISSN 0734-2071. doi: 10.1145/1365815.1365816. URL http://doi.acm.org/10.1145/1365815.1365816.

[38] Cockroach Labs. Serializable, lockless, distributed: Isolation in cockroachdb. https://www.cockroachlabs.com/blog/serializable-lockless-distributed-isolation-cockroachdb/, 2018.

[39] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008. doi: 10.1145/1454159.1454167. URL http://dx.doi.org/10.1145/1454159.1454167.

[40] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pages 251–264, Hollywood, CA, USA, October 2012. Usenix. URL https://www.usenix.org/system/files/conference/osdi12/osdi12-final-16.pdf.

[41] DATASTAX. Configuring data consistency in cassandra. http://docs.datastax.com/en/archived/cassandra/2.0/cassandra/dml/dml_config_consistency_c.html, 2018.

[42] DataStax. How are Cassandra transactions different from RDBMS transactions? https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlTransactionsDiffer.html, 2018.

[43] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Symp. on Op. Sys. Principles (SOSP)*, volume 41 of *Operating Systems Review*, pages 205–220, Stevenson, Washington, USA, October 2007. Assoc. for Computing Machinery. doi: http://doi.acm.org/10.1145/1294261.1294281.

[44] Phil Dixon. Shopzilla site redesign: We get what we measure. In *Velocity Conference Talk*, 2009.

[45] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Symp. on Cloud Computing*, pages 11:1–11:14, Santa Clara, CA, USA, October 2013. Assoc. for Computing Machinery. doi: 10.1145/2523616.2523628. URL http://doi.acm.org/10.1145/2523616.2523628.

[46] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 173–184, Braga, Portugal, October 2013. IEEE Comp. Society. doi: 10.1109/SRDS.2013.26. URL http://doi.ieeecomputersociety.org/10.1109/SRDS.2013.26.

[47] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. GentleRain: Cheap and scalable causal consistency with physical clocks. In *Symp. on Cloud Computing*, pages 4:1–4:13, Seattle, WA, USA, November 2014. Assoc. for Computing Machinery. doi: 10.1145/2670979.2670983. URL http://doi.acm.org/10.1145/2670979.2670983.

[48] Erlang. disk_log. http://erlang.org/doc/man/disk_log.html.

[49] B. J. Fogg, Jonathan Marshall, Othman Laraki, Alex Osipovich, Chris Varma, Nicholas Fang, Jyoti Paul, Akshay Rangnekar, John Shon, Preeti Swani, and Marissa Treinen. What makes web sites credible?: A report on a large quantitative study. CHI '01, pages 61–68, New York, NY, USA, 2001. ACM. ISBN 1-58113-327-8. doi: 10.1145/365024.365037. URL http://doi.acm.org/10.1145/365024.365037.

[50] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002. ISSN 0163-5700. doi: 10.1145/564585.564601. URL http://doi.acm.org/10.1145/564585.564601.

[51] Grid'5000. Grid'5000, a scientific instrument [...]. https://www.grid5000.fr/, retrieved April 2013.

[52] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. ISSN 0164-0925. doi: 10.1145/78969.78972. URL http://doi.acm.org/10.1145/78969.78972.

[53] Isis Distributed Systems. *The Isis Distributed Toolkit, Version 3.0, User Reference Manuel.*, 1991.

[54] Paul R. Johnson and Robert H. Thomas. The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute, January 1976. URL http://www.rfc-editor.org/rfc.html.

[55] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *Trans. on Computer Systems*, 10(4):360–391, November 1992. URL http://dx.doi.org/10.1145/138873.138877.

[56] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978. URL http://doi.acm.org/10.1145/359545.359563.

[57] Butler Lampson and Howard E. Sturgis. Crash recovery in a distributed data storage system. January 1979. URL https://www.microsoft.com/en-us/research/publication/crash-recovery-in-a-distributed-data-storage-system/.

[58] Kwei-Jay Lin. Consistency issues in real-time database systems. In *Annual Hawaii International Conference on System Sciences*, volume 2 of *Volume II: Software Track*, pages 654–661. IEEE Comput. Soc. Press, 1989. URL http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=48069.

[59] Greg Linden. Marissa mayer at web 2.0. *Online at: http://glinden. blogspot. com/2006/11/marissa-mayer-atweb-20. html*, 2006.

[60] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Symp. on Op. Sys. Principles (SOSP)*, pages 401–416, Cascais, Portugal, October 2011. Assoc. for Computing Machinery. doi: http://doi.acm.org/10.1145/2043556.2043593.

[61] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Networked Sys. Design and Implem. (NSDI)*, pages 313–328, Lombard, IL, USA, April 2013. URL https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final149.pdf.

[62] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: Measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 295–310, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3834-9. doi: 10.1145/2815400.2815426. URL http://doi.acm.org/10.1145/2815400.2815426.

[63] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. The SNOW Theorem and Latency-optimal Read-only Transactions. In *OSDI'16*, pages 135–150, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL http://dl.acm.org/citation.cfm?id=3026877.3026889.

[64] Dan Macklin. Can't afford to gamble on your database infrastructure? why bet365 chose Riak. http://basho.com/bet365/, November 2015.

[65] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *NSDI*, pages 453–468, Boston, MA, 2017. USENIX Association. ISBN 978-1-931971-37-9. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/mehdi.

[66] Moniz, Henrique and Leitão, João and Dias, Ricardo J. and Gehrke, Johannes and Preguiça, Nuno and Rodrigues, Rodrigo. Blotter: Low Latency Transactions for Geo-Replicated Storage. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 263–272, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee. ISBN 978-1-4503-4913-0. doi: 10.1145/3038912.3052603. URL https://doi.org/10.1145/3038912.3052603.

[67] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 479–494, Broomfield, CO, 2014.

[68] Fiona Fui-Hoon Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & Information Technology*, 23(3):153–163, 2004.

[69] Christos H. Papadimitriou. The serializability of concurrent database updates. *JACM*, 26(4):631–653, October 1979. URL http://portal.acm.org/citation.cfm?id=322154.322158.

[70] Sebastiano Peluso, Paolo Romano, and Francesco Quaglia. Score: A scalable one-copy serializable partial replication protocol. In *Proceedings of the 13th International Middleware Conference*, Middleware '12, pages 456–475, New York, NY, USA, 2012. Springer-Verlag

New York, Inc. ISBN 978-3-642-35169-3. URL http://dl.acm.org/citation.cfm?id=2442626.2442655.

[71] Sebastiano Peluso, Pedro Ruivo, Paolo Romano, Francesco Quaglia, and Luis Rodrigues. When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication. ICDCS, pages 455–465, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4685-8. doi: 10.1109/ICDCS.2012.55. URL http://dx.doi.org/10.1109/ICDCS.2012.55.

[72] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Symp. on Op. Sys. Principles (SOSP)*, pages 288–301, Saint Malo, October 1997. ACM SIGOPS. URL http://doi.acm.org/10.1145/268998.266711.

[73] Lin Qiao, Kapil Surlaker, Shirshanka Das, Tom Quiggle, Bob Schulman, Bhaskar Ghosh, Antony Curtis, Oliver Seeliger, Zhen Zhang, Aditya Auradar, Chris Beaver, Gregory Brandt, Mihir Gandhi, Kishore Gopalakrishna, Wai Ip, Swaroop Jgadish, Shi Lu, Alexander Pachev, Aditya Ramesh, Abraham Sebastian, Rupa Shanbhag, Subbu Subramaniam, Yun Sun, Sajid Topiwala, Cuong Tran, Jemiah Westerman, and David Zhang. On Brewing Fresh Espresso: Linkedin's Distributed Data Serving Platform. In *SIGMOD*, pages 1135–1146, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2037-5. doi: 10.1145/2463676.2465298. URL http://doi.acm.org/10.1145/2463676.2465298.

[74] Judith Ramsay, Alessandro Barbesi, and Jenny Preece. A psychological investigation of long retrieval times on the world wide web. *Interacting with computers*, 10(1):77–86, 1998.

[75] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-Monotonic Snapshot Isolation: scalable and strong consistency for geo-replicated transactional systems. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 163–172, Braga, Portugal, October 2013. IEEE Comp. Society. doi: 10.1109/SRDS.2013.25. URL http://dx.doi.org/10.1109/SRDS.2013.25.

[76] Eric Schurman and Jake Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity Web Performance and Operations Conference*, 2009.

[77] Daniele Sciascia, Fernando Pedone, and Flavio Junqueira. Scalable Deferred Update Replication. In *DSN*, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4673-1624-8. URL http://dl.acm.org/citation.cfm?id=2354410.2355159.

[78] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Int. Symp. on Stabilization, Safety, and Security of Dist. Sys. (SSS)*, volume 6976 of *Lecture Notes in Comp. Sc.*, pages 386–400, Grenoble, France, October 2011.

Springer-Verlag. doi: 10.1007/978-3-642-24550-3_29. URL http://www.springerlink.com/content/3rg3912287330370/.

[79] Marc Shapiro, Masoud Saeida Ardekani, and Gustavo Petri. Consistency in 3D. In Josée Desharnais and Radha Jagadeesan, editors, *Int. Conf. on Concurrency Theory (CONCUR)*, volume 59 of *Leibniz Int. Proc. in Informatics (LIPICS)*, pages 3:1–3:14, Québec, Québec, Canada, August 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany. doi: 10.4230/LIPIcs.CONCUR.2016.3. URL http://drops.dagstuhl.de/opus/volltexte/2016/6188/pdf/LIPIcs-CONCUR-2016-3.pdf.

[80] Yongxia Xia Skadberg and James R Kimmel. Visitors' flow experience while browsing a web site: its measurement, contributing factors and consequences. *Computers in human behavior*, 20(3):403–422, 2004.

[81] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Symp. on Op. Sys. Principles (SOSP)*, pages 385–400, Cascais, Portugal, October 2011. Assoc. for Computing Machinery. doi: http://doi.acm.org/10.1145/2043556.2043592.

[82] Basho Technologies. riak_core: Distributed systems infrastructure used by Riak. https://github.com/basho/riak_core, .

[83] Basho Technologies. riak_dt: Convergent replicated data types. https://github.com/basho/riak_dt, .

[84] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society. ISBN 0-8186-6400-2. URL http://dl.acm.org/citation.cfm?id=645792.668302.

[85] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *Trans. on Computer Systems*, 4(2):180–209, June 1979. doi: 10.1145/320071.320076.

[86] Alejandro Z. Tomsic, Tyler Crain, and Marc Shapiro. An empirical perspective on causal consistency. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, pages 2:1–2:3, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3537-9. doi: 10.1145/2745947.2745949. URL http://doi.acm.org/10.1145/2745947.2745949.

[87] Alejandro Z. Tomsic, Tyler Crain, and Marc Shapiro. PhysiCS-NMSI: efficient consistent snapshots for scalable snapshot isolation. In *W. on Principles and Practice of Consistency for*

Distributed Data (PaPoC), London, UK, April 2016. Euro. Conf. on Comp. Sys. (EuroSys), Assoc. for Computing Machinery. doi: 10.1145/2911151.2911166. URL http://dx.doi.org/10.1145/2911151.2911166.

[88] A.Z. Tomsic, T. Crain, and M. Shapiro. Scaling geo-replicated databases to the mec environment. In *Reliable Distributed Systems Workshop (SRDSW), 2015 IEEE 34th Symposium on*, pages 74–79, Sept 2015. doi: 10.1109/SRDSW.2015.13.

[89] Marek Zawirski, Annette Bieniusa, Valter Balegas, Sérgio Duarte, Carlos Baquero, Marc Shapiro, and Nuno Preguiça. SwiftCloud: Fault-tolerant geo-replication integrated all the way to the client machine. Technical Report RR-8347, Institut National de la Recherche en Informatique et Automatique (Inria), Rocquencourt, France, August 2013. URL http://hal.inria.fr/hal-00870225/.